Electrical Modeling of Myocardium and Development of Advanced Pacing Techniques

by

Paul Aaron Belk

Bachelor of Science, Physics
Massachusetts Institute of Technology, 1991

Bachelor of Science, Electrical Engineering
Massachusetts Institute of Technology, 1991

Submitted to the
Harvard-MIT Division of Health Sciences and Technology
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Medical Physics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1998

Author:_____

Harvard-MIT Division of Health Sciences and Technology
April 27, 1998

Certified by:_____

Professor Richard J. Cohen
Harvard-MIT Division of Health Sciences and Technology
Thesis Supervisor

Accepted by:_____

Martha Gray
Co-Director
Harvard-MIT Division of Health Sciences and Technology

Electrical Modeling of Myocardium and Development of Advanced Pacing Techniques

by

Paul Aaron Belk

Submitted to the Harvard-MIT Division of Health Sciences and Technology
on April 27, 1998, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Medical Physics

Abstract

There is substantial interest in developing implantable devices to treat ventricular fibrillation, which is the presumed cause of most sudden cardiac death. Existing implantable devices typically use a high-voltage shock—an energy-intensive and uncomfortable therapy. It would be desirable to develop a device that prevents these arrhythmias using low-energy pacing pulses, but the problem is complex and progress has been slow. Computer modeling can aid this development, but microscopic cell membrane models are too slow and macroscopic cellular automata models have provided too little detail. Recently, techniques have been developed for ensuring that the behavior of a cellular automata model corresponds closely to the behavior of a detailed low-level model of a system.

The objective of the study was to use these techniques to develop a computer model which provides enough information about electrical behavior in the heart to aid in the evaluation of advanced antitachycardia pacing techniques. I used this model to test a hypothetical low-field pacing technique for "stabilizing" a diseased heart against the induction of ventricular fibrillation. Prior experimental studies of a such a technique produced inconclusive results.

Model results suggested that the pacing technique is effective if the pacing field is uniform and the field strength is low, so that only fully-recovered myocardial tissue is excited. When the strength of the pacing field is high enough to capture *partially*-recovered tissue, it is *pro*arrhythmic rather than antiarrhythmic, apparently because of arrhythmogenic interactions with electrical inhomogeneities in partially-excitable tissue. Since real electric fields would be very difficult to control with the necessary accuracy, this is a serious practical limitation of the technique

Thesis Supervisor: Professor Richard J. Cohen
Title: Harvard-MIT Division of Health Sciences and Technology

Acknowledgments

My academic career has been long and difficult, but would have been much more so without the substantial assistance of many people. I will certainly be unable to provide sufficient acknowledgement to them, but I will here try to at least highlight a few of the many contributions others have made to my career and my life.

I will begin and end with my wife Tien. Tien selflessly encouraged me to follow my desire to leave a successful engineering career and become an impoverished student. She then worked to support me financially, emotionally and domestically for nine of the ten years of our marriage. It will take me a long time to make it up to her. Drs. William Whitney, Susan Gardner and John Negele personally championed my case against the wisdom of the MIT admissions department, giving me the chance to prove myself academically and scientifically. The have been of great support since.

For most of my graduate studies, I have been fortunate to have the advice, guidance and support of Professor Richard J. Cohen. Beyond his exceptional understanding of my and many other research fields, he has also constantly demonstrated to me the value of humanity and integrity in research. In four years, he has consistently and patiently given me guidance and correction without ever belittling or patronizing me. He has consistently provided leadership without ever restricting my freedom. He has never once made me feel that my ideas were inferior to his, however obvious it was that they were.

Perhaps my greatest good fortune as a graduate student was the opportunity to work two years with Linda Rosenband. Her intelligence, insight and sheer determination led to the original version of the simulator on which all of my research is based. She was a superb scientific colleague, a good friend, and always a joy to work with. Though Linda left for graduate school nearly two years ago, she can still claim credit for much of what I accomplish, and I still feel her absence.

When I first entered Prof. Cohen's group, I was very fortunate to be able to

depend on the assistance of my colleagues, Antonis Armoundas, Bin He, Ramakrishna Mukkamala, Thomas Mullen, Derin Sherman, and Zhi Hao Yin. I single out in particular my two office mates and friends, Derin Sherman and Tom Mullen. Derin has continuously given me the benefit of an encyclopedic knowledge of both science and technology which is awe-inspiring even by MIT standards. Many difficult technical obstacles have vanished during brief conversations with Derin. Tom has helped me with every aspect of my research and with nearly every other aspect of my life in the lab since the first day I arrived. I have relied on his knowledge, understanding and assistance each day since, and he has proven to be the most dependable colleague I have ever had. It has been a privilege to work with him on any project, small or large, and I hope to have many more opportunities.

Throughout my study of the physical properties of the heart, I have been guided by Dr. Yuri Chernyak. Yuri has given me the benefit of his deep and comprehensive understanding of mathematical physics, of how to apply that understanding to biological systems, and of his passion for partial differential equations; and he has done it all with patience and humor. Dr. Andrew Feldman has been my main collaborator, instructor, and goad since he completed his Ph. D. last year. Without his skills and guidance, and without the benefit of his vast knowledge of the field, I would still be wandering in the wilderness.

Dr. John Triedman was very generous with help and guidance. He helped teach me the way I had to view modeling for it to have any meaning to real-world electrophysiology and helped make sure that my model was always grounded in reality. He was also very generous with time and resources, even in areas not directly related to my thesis. Without his kindness my last year would have been much more difficult. Prof. Roger Mark was indispensible in helping to define my research objectives and conclusions, and his scientific challenges constantly helped to refine my work. I learned much more for having each of them on my committee.

Throughout my studies I have drawn on the assistance of my brother Nathan, who is not only a Ph. D. in physics, but also an expert in both electronic and experimental design. My brother David is an M. D., in addition to being an expert in physiology. I leaned heavily on him through medical school and since, and as my knowledge of medicine grows, so does my amazement at his. Our mother has provided each of us immeasurable support throughout our long, tortuous and often seemingly-hopeless journeys through academia. I would also like to thank my friends Stephanie Nonas and Yuqi Han for having the patience to provide me with much-needed perspective through the end of my dissertation.

Most of my research has been on computers and has benefitted from an extraordinary array of available software tools. Without exception, these tools were provided for my use free of charge by superb programmers who had no direct stake in my work. More than reducing the cost of my research, they provided me the freedom to experiment with new ideas without first having to justify the purchase of new software—an essential ingredient in research. I am therefore deeply grateful to Linus Torvalds and the Linux Community, Richard Stallman and the Free Software Foundation, Knuth, Lamport, Boossens, Mittelbach, Samarin and the TEX community (special gratitude to Michael Piefel for `lgrind` and personal assistance), Larry Wall for Perl, and too many others to name.

And in the end, my greatest gratitude again to Tien. She supported me to the end—beyond what I thought anyone would have endured. And of all the wonderful things she has given me, the most wonderful of all is our son John. John, who sees the end of my graduate work, was named for my father, who saw the beginning, though they never met. I therefore dedicate this work, and very much more, to them.

To John Belk

Contents

List of Figures

List of Tables

Chapter 1

Motivation and Background

Humanity has a long-standing fascination with the heart. It is a mechanical system that operates continuously throughout life, though we are only occasionally aware of it. Evidence of its constant operation, the palpable heartbeat and the sounds of valve closings, can be inspected by any curious layperson at almost any time, but its operation is usually obvious only in times of stress. It seems to respond in general to our emotional state, but is otherwise beyond our conscious control.

An investigator curious about the heart will be amply rewarded for his effort. The heart is an extraordinary mechanical system—a high-output pump which performs reliably approximately every second throughout life. As such, the heart operates with a durability and reliability which few, if any, human-designed systems even approach. This is of great interest to engineers. The heart is also an extraordinary electrical system. Heart tissue forms a distributed three-dimensional system of *excitable media*, meaning that it is a diffusively-coupled medium that also exhibits wave behavior. This links the properties of heart tissue to such diverse processes as the Belousov-Zhabotinskii chemical reaction [13, 75], and the propagation of flame [74]. The mathematical structure of such systems is an area of great interest to physicists.

Equally fascinating, however, has been our understanding from ancient times of

the instantaneous relationship between the heartbeat and life. We have long been aware that the loss of heartbeat was the immediate equivalent of death. Indeed, until the last few decades, the loss of the heartbeat was the *definition* of death, and even now this is the definition most often used.[1] Modern medicine has provided an understanding of the fact that the cessation of cardiac function may sometimes *result from* death rather than actually *cause* death but, especially in the developed world, cardiovascular disease remains the single largest cause of mortality, accounting for 42.7% of all deaths in the United States in 1991 [9]. Of these, *sudden cardiac death*, abrupt loss of heart function, accounted for approximately 250,000 deaths [10]. Medical research on the cardiovascular system is most often driven by the enormous health impact of these diseases.

Current understanding of the cardiovascular system has demonstrated that the problems in medicine associated with heart disease are inextricably linked to the problems in physics of understanding of the propagation of electrical impulses through cardiac muscle. Sudden cardiac death is known to be an acute aberration of the propagation of the excitation wavefront which is responsible for the heart beat. Predicting this aberration, and understanding how to treat it, leads one back to a study of the basic physical properties of cardiac tissue that control the propagation of these excitations.

In this thesis I will present a study of the electrical properties of the heart. This study is driven equally by the desire to understand the basic physics of those properties, and by the desire to develop useful clinical interventions for the cases when disease leaves them inadequate to reliably support the heart beat. My approach will be the development of a computer simulation of the electrophysiology of the heart

[1] There are notable exceptions on both sides of the definition. "Brain death" is a recognition of the fact that broad physiologic viability, including a functioning cardiovascular system, is an incomplete definition of life in the absence of high-level central control. Thus a person with a heartbeat may be "dead." On the other side, "sudden cardiac death," the acute loss of the heartbeat, can be reversed using modern medical technology, so that a person without a heartbeat may not yet be dead or, more properly, the person is dead, but not fatally so.

which provides sufficient detail to accurately model the induction and termination of lethal electrical disturbances in the diseased heart, while being efficient enough to allow rapid whole-heart simulations of proposed interventions.

I will first use this model to explore the detailed physical processes that allow the development of lethal arrhythmia in a diseased heart. I will use that understanding to explore a technique for pacing with diffuse fields at low energies to stabilize the heart against lethal arrhythmia, by preemptively capturing large regions of myocardium before arrhythmia can develop. This technique is based on an experimental study performed 1992, which paced at multiple sites on the ventricle [17]. The results of this study were inconclusive and I therefore hope to use the model to study the efficacy of the approach.

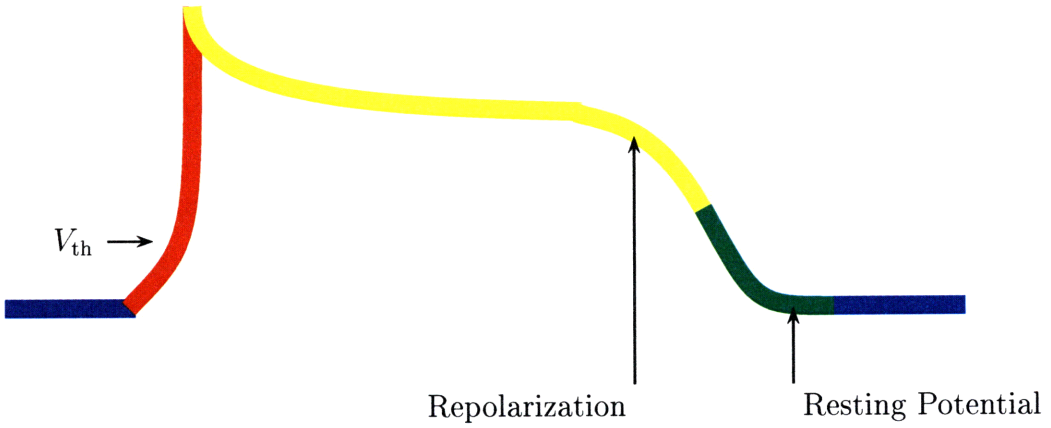
1.1 Cardiac Electrophysiology

Rapid progress in understanding of the intrinsic control of the heart began with Einthoven's discovery at the turn of the century that the palpable heart beat was directly associated with the movement of small currents at the surface of the skin. This discovery formed the basis of the modern science of cardiac electrophysiology. Electronic technology quickly improved to make the measurement of these currents (actually, the potentials associated with the currents), the *electrocardiogram* (ECG) routinely measurable with electrodes placed on the skin. For many decades, these measurements have been an essential part of the non-invasive evaluation of any patient with suspected cardiac abnormalities.

1.1.1 The Action Potential

The development of the ECG lead to the understanding of the heart as an electro-mechanical system. It is now understood that the response of all neuro-muscular

Figure 1-1 Schematic of the Cardiac Action Potential. The colors represent the functional "state" of the myocyte: blue is full excitability, red is active propagation, yellow is the refractory period, and green is partial excitability. This is the color scheme that will later be used in simulation snapshots.



tissue is characterized by an electrical "action potential" propagating along the cell membranes of that tissue. In nervous tissue, this action potential provides for the conveyance of signals to and from the central nervous system. In muscle, the action potential evokes substantial changes in the intracellular Ca^{++} concentration which activates the molecular contraction machinery in the cell. The cardiac action potential refers to the evolution of the transmembrane voltage of a cardiac myocyte after it has been electrically "stimulated." It is useful to review the most important features of the cardiac action potential (Fig. 1-1):

resting potential (Blue in Fig. 1-1) For a non-automatic cardiac myocyte (a myocyte that does not spontaneously activate), the only stable electrical state for the membrane is with a voltage of approximately -90 mV across the membrane, with the extracellular potential taken to be 0. When the transmembrane voltage of a healthy cardiac myocyte is perturbed, it will always return to this value.

threshold voltage (V_{th} in Fig. 1-1) If the transmembrane potential of a healthy

cardiac myocyte is raised above its threshold voltage it will undergo a full excitation response (a full action potential). If the transmembrane potential is perturbed, but not driven above the threshold voltage, it will return quickly to the resting potential. The threshold voltage is generally approximately 30 mV above the resting potential, but can change depending on the external ionic environment and the internal state of the myocyte.

rapid upstroke (Red in Fig. 1-1) This is commonly referred to as "phase 0" of the cardiac action potential. On excitation, the transmembrane potential spontaneously increases to approximately 30 mV. This occurs very rapidly, over a period of approximately 3 msec.

refractory period (Yellow in Fig. 1-1) The period during which a myocyte cannot be stimulated. There is a brief period of rapid repolarization after the upstroke ("phase 1") and a plateau, usually lasting over 100 msec, during which the trans-membrane voltage changes very little ("phase 2"). The beginning of repolarization ("phase 3") is marked but does not correspond to the beginning of excitability.

Repolarization (Period between "Repolarization" and "Resting Potential" on Fig. 1-1). The transition between "depolarization" and "repolarization" ("phase 3"). This period is responsible for the "T-wave" in the ECG and is usually much longer than the rapid upstroke.

relative refractory period (Green in Fig. 1-1) Myocytes begin to regain excitability when repolarization ("phase 3") is partially completed and the fast Na^+ channels begin to reactivate. Since these channels take time to recover, the population available for an excitation response increases continuously during this period, resulting in subnormal propagation speed of excitation waves. Even when the trans-membrane voltage has returned to its resting value, the myocyte

is still not fully recovered because of the time delay in reactivation of the Na^+ channels.

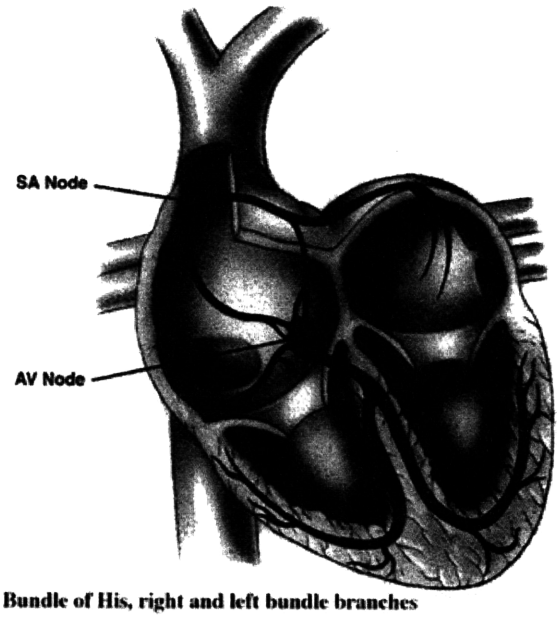
1.1.2 Propagation of the "Heartbeat"

Unlike the other rapid-response muscle systems in the body, the skeletal muscles, the electrical control system in the heart is nearly entirely self-contained. Like the heart, the skeletal muscles are capable of contraction and relaxation cycles well in excess of 1 Hz, but control of skeletal muscle is entirely central. Each skeletal muscle fiber will "fire" (develop an action potential and contract) only in response to the firing of the axon coupled to that fiber's motor end plate. The firing of that axon is exclusively controlled by the central nervous system.

To understand the uniqueness of the heart as a muscle system, it is interesting to consider the role of the heart in the developing organism. A centralized system would probably be impractical for essential control of the heart. Development of a functional central nervous system implies substantial sophistication of the organism, and such a high level of complexity would require the foundation of a functional circulatory system. From both an evolutionary and an embryological standpoint it therefore seems unlikely that any system controlling the heart could await the development of a functioning central nervous system. Control of the heart must therefore clearly begin almost entirely independently, and it is hard to envision a scenario whereby the functioning autonomous control would later spontaneously "switch" to central control.

The resulting autonomous control of the heart is breathtakingly elegant. In a normally-functioning heart the cardiac muscle cells ("cardiac myocytes"), perform both the contractile task of a normal muscle cell and the control task of a nerve cell. There are three relatively-small collections of cardiac muscle cells which can spontaneously generate periodic action potentials: the *sinoatrial node* near the entrance of

Figure 1-2 Electrical Anatomy of the Heart [10]



the superior vena cavea in the right atrium, the *atrioventricular junction* near the annulus of the tricuspid valve in the atrioventricular grove, and a region in the ventricular bundle of HIS (see Figure 1-2). Once an action potential is initiated in one of these three locations, an excitation wavefront propagates through normal myocardial tissue, along the membranes of the cardiac myocytes themselves, and through specialized conduction tissue, the HIS/purkinje system (Fig. 1-2). The depolarization wavefront propagates with power gain through a highly non-linear medium, but the propagation is entirely "autonomous," i.e., there is no mechanism for interrupting or directing the wavefront once initiated.

The fact that the heart is self-contained provides a great deal of robustness to the system, allowing, for example, proper cardiac function following complete central denervation as after a transplant or advanced neuropathy. A patient with advanced debilitating disease rendering their skeletal muscles–even their respiratory muscles–

useless, may still have almost normal cardiac function.

This autonomy comes at a price, however. If the heart, which is a comparatively simple device neurologically, begins to malfunction electrically, it is impossible for the central nervous system to intervene in any effective way. This is the basis of the mortality and morbidity resulting from cardiac arrhythmia—electrical disturbances within the heart cannot be reversed from outside the heart. This is critical for understanding the electrical nature of the heart:

> proper electrical function of the heart depends *entirely* on the proper autonomous propagation of the depolarizing wavefront.

If proper propagation is disturbed by intrinsic disease or by an external event, there is no central mechanism for restoring normal electrical function to the heart. Thus the very elegance of design that makes the circulatory system possible also contains one of it's greatest vulnerabilities: the lack of any mechanism for restoration of proper electrical function once the electrical propagation pathways in the heart have been disturbed.

1.2 Electrical Abnormalities

Disturbances in cardiac control, or *rhythm disturbances* ("arrhythmias," "dysrhythmias") can broadly be divided into two categories: *tachyarrhythmias*, where the heart beats too fast, and *bradyarrhythmias*, where the heart beats too slowly. Both types of conditions can range from benign to life-threatening, and both types are amenable to medical intervention, either pharmacological or electronic. Bradyarrhythmias are often the result of a failure either of one or more groups of autonomous pacemaker cells (the sinoatrial node and the automatic portion of the atrioventricular junction), or part of the conduction system (usually the atrioventricular node), though clinical bradycardia can be a healthy conditioning response in athletes with well-trained car-

diovascular systems.[2] When bradycardia is a response to conditioning, it is often due to a change in the "setpoint" of the autonomic nervous system which influences heart rate.

Treatment of bradycardia is often accomplished by the implantation of an electrical device called a *pacemaker* which provides periodic electrical stimulation of a section of heart tissue through a small pacemaker electrode. Since the underlying problem in these cases is often either an insufficient stimulation rate from the intrinsic pacemaker cells, or an electrical obstruction which prevents intrinsically-generated stimuli from propagating, the pacemaker substitutes for these stimuli.

It is interesting to note that for disease in which the intrinsic pacemaker nodes are functional but have a spontaneous frequency which is too low, the artificial pacemaker can *completely* supplant the function of these cells, that is, all electrical stimulation comes from the artificial pacemaker. This is because the electrical impulse from the artificial pacemaker causes the intrinsic pacemaker cells to reset to phase 0 in their internal depolarization cycles and, since they would not then fire until after the next impulse generated by the artificial pacemaker, they are effectively suppressed by the artificial pacemaker, a mechanism called *entrainment.* This is an illustration of a general principle in cardiac electrophysiology:

> In a region of the heart without significant electrical obstruction, the firing rate is set by the pacemaker which has the highest frequency. All lower-frequency pacemakers are entrained.

This principle is important for understanding of tachyarrhythmias.

Tachyarrhythmias can sometimes be the result of inappropriate automaticity in a group of myocytes, either intrinsic pacemaker cells firing too frequently or non-pacemaker cells developing automatic behavior. *Sinus tachycardia* is a condition in which the automaticity of the sinoatrial node is inappropriately enhanced, resulting

[2]Even the author was briefly bradycardic in the period before he began work on his thesis.

in tachycardia. A more serious condition, *ectopic tachycardia*, results when a region of the ventricle or atrium which is not normally automatic, become automatic at a higher rate than the intrinsic pacemaker nodes. Both these types of tachycardia are normally treated with pharmacological agents that decrease the automaticity of the cells involved, though incessant ectopic tachycardia can also be treated by ablation of the ectopic focus. In either of these cases, an artificial pacemaker would clearly be an ineffective treatment since it could not, in general, entrain the other cells without beating faster and thus exacerbating the problem.

Tachycardia can also result from normal automaticity coupled with aberrant *propagation* of electrical depolarization. Such conditions are referred to as *reentrant tachycardia*, because the depolarization wavefront *reenters* part of it original pathway and thus becomes self-sustaining. These types of arrhythmias can result in hemodynamic compromise when they cause a heart rate which is too fast to allow sufficient diastolic filling. More importantly, when reentrant activity becomes disorganized, with several reentrant pathways simultaneously active, distant regions of the heart lose synchrony. Without synchrony, pumping action in the effected chambers ceases. This condition is called *fibrillation* and can occur either in the atria or the ventricles.

Atrial fibrillation (AF) is a sometimes debilitating condition resulting in decreased cardiac output from a rapid, irregular heartbeat with ineffective atrial activity. It also creates large regions of hemostasis in the atria, often predisposing the patient to atrial thrombi and thrombo-embolic events such as stroke. However since the ventricles can continue to beat almost normally, AF is usually not immediately life-threatening. *Ventricular fibrillation* (VF) is usually the proximate cause of *cardiac sudden death.* Loss of consciousness is immediate and in the absence of medical intervention, death results within five minutes. Approximately 250,000 people die each year in the United States from cardiac sudden death [10].

In some cases, it is possible to terminate low-frequency reentrant tachycardia with

pacing impulses timed to fall between recovery and reexcitation of the reentrant wavefront. This makes reexcitation impossible and therefore terminates the tachycardia. The only effective treatment for ventricular fibrillation, however, is *countershock*, in which a large electric shock is applied to the heart to simultaneously excite all (or most) myocytes, rendering further propagation of depolarization wavefronts impossible for at least 25–35 msec[3] [67]. If successful, this allows normal electrical activity to resume. Countershock is also used to terminate reentrant tachycardia that is not susceptible to pace termination.

There are two ways in which a patient in VF might hope to receive countershock. The shock can be administered through external defibrillator paddles by trained emergency personnel. This requires of course, that the patient be within a few minutes of the equipment and personnel when the episode occurs, and the necessary people are notified in sufficient time to deliver the therapy. This problem can be overcome by implantation of an implantable cardioverter/defibrillator (ICD). These devices are very costly, however, and are only implanted in patients shown to be at high risk for lethal tachyarrhythmia. Since a large number of victims of sudden cardiac death have no prior symptoms, this is clearly inadequate.

Intense medical research is being directed to a more detailed understanding of tachyarrhythmias. Among the the motivations for that research are the potential for improved screening tests to identify patients at risk for developing life-threatening tachyarrhythmias, and the potential for improving the therapies for treating those arrhythmias. As it becomes possible to detect more patients at risk for possibly lethal tachyarrhythmias, it will become more desirable to reduce the cost of implantable devices for controlling those arrhythmias. One approach to doing this would be to decrease the energy requirements of the intervention by developing a lower-voltage

[3]The refractory period after countershock or high-strength pacing impulses depends strongly on when the shock is delivered relative to the internal state of the cell. When the shock is delivered during the refractory interval, excitation of the cell does not occur and there is no "action potential", but the refractory period is extended [35,67]

alternative to countershock.

1.3 Modeling

A *model* is any system with properties sufficiently similar to another system under study that the behavior of the model system can be examined to provide understanding of the system under study. Models may in general be experimental of theoretical. In medicine there are a large variety of models used. *Animal models* are usually experimental investigations in which animal physiology is studied to provide insight into comparable human physiology. Often the researcher attempts to create conditions in an animal which are believed to mirror a particular human disease process such as myocardial ischemia or hypertension. Animal models have the advantage that true physiologic processes can be studied and data can be gathered using techniques that would be unacceptable in humans. These facts make animal models indispensable in medical research because they still provide the closest approximation available to human physiology and data gathered from animals is much less susceptible to erroneous or incomplete assumptions by researchers.

The similarity of animal physiology and disease processes with the human physiology is limited by two factors. First, the physiology of an animal is never identical to human physiology and it can be difficult to assess the importance of the differences. Second, the human disease processes under study often evolve over many years and it may be impractical or impossible to completely reproduce the effects of these processes in animals.

These problems can be clarified by simple examples. Consider first the problem of differing physiology between humans and animals. Generally, mammalian hearts are studied as models for human hearts. A rat heart is a convenient animal model because rats can be raised quickly and easily, and experiments on rats do not re-

quire sophisticated medical facilities. Rat hearts produce action potentials that are qualitatively similar to human action potentials, but their ventricles are too small to support anything similar to ventricular fibrillation. Working with larger mammalian models, such as canine and swine models is substantially more difficult, but these models allow the production of similar arrhythmias to those seen in humans. Even in these animals, however, the quantitative nature of the action potential is different, so it can be difficult to interpret data on, for example, the relationship between action potential duration and diastolic interval. Such a quantitative relationship can be very important in studying the details of induction and termination of arrhythmia.

To this problem is added the problem of modeling disease. It has long been suggested that one of the major contributing factors to the induction of ventricular fibrillation is spatial variation in the time required for cardiac tissue to recovery excitability after excitation (*refractory time*) [7, 8, 37, 45, 51, 52, 59, 63]. This is called the "Dispersion of Refractoriness" hypothesis. This dispersion may result from acute or chronic ischemia due to coronary artery disease, or to other types of myocardial disease. Many techniques are used to create a presumably-similar pattern of dispersion in animals: for example, acute ischemia due to ligation of coronary arteries [59], change in local potassium concentration [51], injection of cardiotoxic drugs [7] and hypothermia [62]. These models all give useful results, but part of the interpretation of the results must always be a consideration of whether the physiologic conditions induced by such acute experimental interventions accurately reproduce what is in humans very probably the result of a chronic disease process. Further one must consider whether the desired result, dispersion of refractoriness, is the *only* significant result of the intervention, or whether other physiologic factors, such as excitability are different between humans and models.

Theoretical models are different from experimental models in many respects. A theoretical model always begins with a conceptual understanding of the system on

the part of the researcher. That understanding is then incorporated into the design of a system that is often completely different from the system under study. One may choose, for example, to create a theoretical model of the hemodynamics of the cardiovascular system beginning with an understanding based on compliance and resistance of blood vessels and the time-varying properties of heart chambers. These concepts may then be mapped to similar electronic circuit elements, and simulated on digital computers [14, 21] or even implemented as actual circuits.

Such a hemodynamic model of the cardiovascular system represents a great simplification from the actual physiology. There are obvious disadvantages to such a simplification, but there are also very important advantages. Because the model operates in a parameter space that is substantially reduced from the physiologic system, it is possible to collect and examine data very efficiently, focusing only on those variables that are of direct interest. The simplification also allows simulated experiments where important parameters, such as arterial compliance or resistance, are altered without having to worry about other physiologic variables that would be affected by such alterations. Finally, and perhaps most importantly, such a simplified model provides for a *test* of the understanding of the system: if the assumptions (understanding) from which the model was constructed are correct and sufficient, then the model behavior should correspond closely with observed physiologic behavior. Thus theoretical modeling is an iterative process in which understanding is progressively tested and refined through implementation of models.

The above arguments apply equally well to electrical models of the heart. In analogy to the elegant and simple hemodynamic model of the cardiovascular system, some of the earliest models of the heart were designed to encapsulate the electrical behavior of the heart using a technique called *finite-state cellular automata* [52]. This technique begins with the observation that myocardial tissue can be characterized by an *excitation threshold* and *excitation response* (Sec. 1.1.1) The two most significant

aspects of this *excitation response* were

1. The membrane remained *refractory* to subsequent excitation for a period of time called after excitation (the *refractory period*); and

2. The excitation response of the membrane created sufficient current to excite neighboring myocytes, that is, excitation was self-propagating.

Using finite-state cellular automata, discrete sections of myocardial tissue are modeled as individual elements that excite in response to the excitation of neighboring elements. Because such systems display only the most essential wave behavior in myocardial tissue, they provided some of the earliest tests of hypotheses of macroscopic behavior in the heart, including development of arrhythmia (Sec. 1.4.3).

Many different types of models can be constructed to represent electrical propagation in the heart. Some of the most detailed models are designed to reconstruct the behavior of individual ion species with the cell membrane. These models are useful for understanding the behavior of the cardiac action potential, especially in response to changes in ionic environment or channel activity (Sec. 1.4.1). Other models seek only to model net current flow across the membrane without concern for the individual ionic currents. These models are most useful for understanding relationship between macroscopic excitation wave behavior and microscopic membrane current-voltage (I-V) relations, because they condense the intricate relations between ion species and individual channels (Sec. 1.4.2).

1.4 Computer Models

The most common method for implementing theoretical electrophysiologic models is on digital computers. All three of the model types described above, ion channel models, membrane current (PDE) models, and finite state cellular automata have

been implemented on computers by many researchers, over the past thirty years. In this section, I will provide a brief description of assumptions and the strengths of implementation of each of these model types. This will provide background for understanding the modeling approach used in this study.

1.4.1 Ion-channel Models

Electrical activity of all neuromuscular tissue is based on the ability of the cell membranes in this tissue to selectively control the passage of specific ions. This relationship was first described quantitatively for the membrane of the squid axon by Hodgkin and Huxley [38], for which they were awarded the Nobel Prize. In these models, specific membrane ion channels are identified, and channel conductance is empirically determined as a function of all parameters in the model. The membrane voltage is continuously determined by integrating the ion current through the channels [11, 46–48]. This approach provides the most detailed model of the membrane electrical activity (Figure 1-3).

A variation of this approach, where the activity of the channels is modeled as a Markov process, has been implemented in the laboratory of Prof. Cohen [73]. Although this formulation is somewhat more elegant and tied to more intuitive physical mechanisms, it remains far too complex computationally to simulate physiologically-significant volumes of heart tissue.

1.4.2 Net Current Models

A second approach models myocardial tissue as a continuous excitable medium governed by relatively-simple reaction-diffusion equations [28, 54]. These models ignore the complexity resulting for the variety of ion species and ion channels in the membrane and focus instead on the *net* i-v relationship for the membrane. This provides a vast reduction in the number of independent parameters in the problem and there-

Figure 1-3 Excerpt of Equations for Luo-Rudy ion channel model [46]

$$
\begin{aligned}
\frac{dV}{dt} &= -\frac{I}{C_{\text{membrane}}} \\
I &= \sum_i I^i_{\text{Na}} + \sum_j I^j_{\text{K}} + \ldots \\
I_{\text{Na}} &= am^3hj(V - E_{\text{Na}}) \\
\frac{dm}{dt} &= \alpha_m\,(1 - \beta_m\,y) \\
\alpha_m &= \frac{0.32(V + 47.13)}{1 - e^{[-0.1(V+47.13)]}} \\
\beta_m &= 0.08\exp(V/11) \\
\frac{dh}{dt} &= \alpha_h\,(1 - \beta_h\,y) \\
\alpha_h &= 0.134\exp\left(-\frac{80 + V}{6.8}\right) && (\text{for } V < -40\,\text{mV}) \\
\beta_h &= 3.56\exp(0.079V) + 3.1 \times 10^5\exp(0.35V) && (\text{for } V < -40\,\text{mV}) \\
&\vdots
\end{aligned}
$$

fore can give very elegant models. These models have been quite successful in testing hypotheses for mechanisms of dysrhythmogenesis [56, 66]. In particular, they have shown the increased vulnerability of the heart during the *relative refractory* period, which is due directly to decreased excitability.

These models are sufficiently simple to be nearly-tractable computationally, even on whole hearts, but because of the tight coupling between different aspects of the tissue (for example, the difficulty in changing refractory period without also changing propagation speed), it remains difficult to model specific disease states assumed to lead to dysrhythmia.

1.4.3 Cellular-automata Models

Representations of whole-heart (or at least whole-chamber) electrical phenomena have been carried out for more than 30 years [52] using the technique of finite-state cellular automata. Such models frequently seek to represent only the abstract macroscopic aspects of a system, at the expense of microscopic representation. Such representations have often clarified understanding of macroscopic phenomena [52,63]. However, often as a result of insufficient attention to microscopic phenomena and to underlying physical principles, simulation results have been misinterpreted as demonstrating cardiac electrical phenomena [33, 40] which were in fact merely simulation artifact [27, 73]

Macroscopic cardiac electrical simulators, especially finite-element simulators, have usually been quite good a incorporating observed physiologic behavior, especially as it relates to the details of recovery of the action potential. They have often placed little emphasis, however, on representation of basic physical law, such as symmetry of propagating wavefronts and detailed relationships between wavefront curvature and propagation speed, which are basic to all wave phenomena.

The approach of constructing a model based on a minimal set of states and rules is generally very useful for testing model hypotheses, i.e., for determining whether the

Table 1.1 States used in the first finite-state cellular automata model of cardiac conduction [52]

State	Characteristic
RESTING	Element is fully excitable at nominal propagation speed
EXCITED	Element has depolarized in last time step and is capable of depolarizing neighboring elements
ABSOLUTE REFRACTORY	Element cannot be excited
RELATIVE REFRACTORY 1	Element can be excited but only at 25% nominal propagation speed
RELATIVE REFRACTORY 2	Element can be excited but only at 33% nominal propagation speed
RELATIVE REFRACTORY 3	Element can be excited but only at 50% nominal propagation speed

postulates of a model are *sufficient* to account for observed behavior. For a system as complex as myocardial tissue, this provides the ability to disregard those details of system behavior which do not contribute significantly to a particular phenomenon, and therefore to aid in understanding how the phenomenon may be manipulated.

The first CA model of cardiac excitation by Moe, Rheinholdt, and Abildskov in 1964 [52] chose only states representing rest, initial excitation, absolutely refractoriness, and three states of partial refractoriness (Table 1.1). The duration of the EXCITED state (representing the action potential duration) was taken to be variable in space and the model demonstrated fibrillatory-like behavior. An even simpler model by Smith and Cohen in 1986 [63] reduced the number of states to two: RESTING and ABSOLUTE REFRACTORY, and demonstrated not only fibrillatory-like behavior in a ventricular model but also demonstrated 2:1 and 3:1 conduction block, short runs of reentrant dysrhythmia, and electrical alternans.

Cellular-automata models have received extensive use for other studies as well. They have been used to study theories regarding the nature of *Torsade de Pointes* [1, 2], initiation of spiral waves [4], simple anti-tachycardia pacing regimes [3], and

evaluating the information contained in surface electrograms [72]. Often they have provided information about the effects of excitation wavefront propagation that would have been difficult to obtain experimentally.

Such models are limited, however, when one attempts to take them beyond the test of *sufficiency* to direct *analysis of physiologic behavior.* Analysis represents a qualitatively distinct purpose of modeling: to provide a controllable model of detailed system behavior. There have been several notable attempts to analyze complex detailed behavior of excitation wavefronts in disorganized states such as ventricular fibrillation, and the results of these models have occasionally been misunderstood due to simulation artifact. Two of the most notable such cases occurred in 1990 [33], and 1991 [40], when two separate finite-element models reported the degeneration of organized spiral waves into disorganized fibrillatory patterns. In both cases, it was possible subsequently to prove that the result was due to symmetry violations imposed on the models, rather than to the intrinsic behavior of the wavefronts [27, 73]. For analytic work, therefore, it is clear very important to be concerned with faithfulness of physical representation.

1.5 Choice of Model

The major issues determining the choice of a model are almost always a combination of practical and abstract considerations. Even in a hypothetical situation in which any model could be practically implemented, the choice of model would depend on the goals of the study. In this study, I use the following criteria to determine the suitability of a model:

1. **Practicality.** It must be possible to implement and use the model given the technology and resources currently available. For this study, it is clearly necessary to implement a model in which the development of an arrhythmia on a

human-size ventricle takes only a few minutes on a computer workstation. Due to the exponential progression of computer power, the impact of this requirement changes rapidly[4].

2. **Reliability.** An ideal implementation of a model would provide a complete replacement for animal (human) experiments. While this is obviously not a practical goal, it provides one basis for evaluation of such models: *an analytic or computational model of a system is useful only to the extent that it decreases, to some extent, reliance on other models.*

3. **Understanding.** Even a completely faithful model of physiology may not be very useful, because the goal of a study is not simply to determine *whether* a behavior occurs but *why* that behavior occurs. One of the greatest potential benefits of modeling is the ability to get a clearer picture of a process than would be available during an actual physiologic study.

Although it is obviously necessary to satisfy all the above requirements, in many ways it is requirement 3 that is the most critical and the most important in the success of this study. We will examine the models described above with respect to all of these requirements.

1.5.1 Practicality

Consider first requirement 1, practicality. For animal models, there is obviously no problem. The only type of model that this requirement absolutely rules out is an ion-channel model. Because of the sharpness of some of the features of the cardiac action potential (e.g. the rapid upstroke), the duration of other features (e.g. the plateau), and the coupled non-linear differential equations relating them, an attempt to model

[4]In the time since I began this study it has become possible, based on this requirement, to implement a three-dimensional model of the ventricle on a workstation costing approximately $10,000.

a large enough region of heart tissue to support anything approximating macroscopic behavior given foreseeable computer resources will not be practical in the near future. A crude measure of this requirement is the time it takes a "reasonable" implementation of a model on a workstation to simulate 1 second of physiologic response of one square centimeter of tissue. For a Luo-Rudy type formulation, one second of physiologic response requires approximately 5000 minutes of workstation time per square centimeter[5]. By this measure, even a few seconds of sustained arrhythmia on a human-size ventricle would take months.

Net-current models (Sec. 1.4.2) are nearly practical. Using the crude measure of efficiency described above, one second of physiologic response on a net-current model would require about 30 seconds of workstation time per square centimeter. This is still not practical on a workstation such as I use in my study, but efficient implementations of the partial differential equation formulations used by Starobin [66], Pertsov [56] and others, running on supercomputers, have been used to study arrhythmogenesis.

Cellular automata models provide the most practical implementations. Depending on the state transition rules and element size chosen they can be used to model human-sized ventricles and produce 1 second of physiologic data in only a few seconds of computer time.

1.5.2 Reliability

Requirement 2, reliability, can be considered more important that the requirement of practicality. A high-speed implementation of an oversimplified model will only rapidly contribute to confusion, especially if the limitations of the model are not well-understood.

There is no completely reliable model. Even a particular human heart may be

[5]This is obviously a *very* crude measure. A skilled implementer could conceivably reduce this number by a factor of 10. It is still clear, however, that what should require minutes would require weeks.

substantially different from the "norm," rendering the study invalid. Nevertheless, animal models, especially in large mammals, are usually taken as the gold standard because any working heart necessarily duplicates the most important features of any other working heart.

Ion channel models are generally considered more reliable than other models because they are designed to reproduce the detailed microscopic behavior that is directly responsible for the propagation of excitation waves in the heart. Ion-channel models are also the *only* models available which are reliable for the study the cardiac action potential itself, since these are, be definition, the only models which are designed to reproduce the action potential in detail.

The reliability of ion channel models is limited, however, by the fact that their design is based on the observation and identification of specific ion channels, rather than of the entire cardiac action potential. Ion channel models are based on an empirical understanding of the ion channels themselves, rather than on large-scale behavior. This means that these models necessarily model the effects only of those channels which are experimentally identified, rather than being driven by an attempt to faithfully reproduce large-scale electrophysiology.

Net current models make no attempt to reproduce the details of the action potential, as do ion channel models, but rather to reproduce the macroscopic wave phenomena. Compared to ion channel models, therefore, net current models are less reliable for study of the details of the action potential, but *more* reliable for the study of the macroscopic behavior that leads to arrhythmia. The partial differential equations used in these models are synthesized from an understanding of the overall behavior of the membrane i-v relationship. The equations are therefore limited not only by the understanding of the relationship, but also by the ability to express that relationship in terms of partial differential equations (the ability to "fit" observed behavior to a source term for the PDEs). These models have been most useful in

studying the types of qualitative wave behavior excitable media support.

Cellular automata models have thus far been the *least* reliable models of cardiac electrophysiology. This is due to the fact that the transition rules which completely specify the behavior of these models may be chosen completely *ad hoc*. Since these rules had not been forced to correspond to the physical system, or even systematically tested for correspondence, it is not surprising that some of the behavior observed in these models is solely due to the limitations of the models themselves [27, 33, 40].

Techniques have recently been developed to constrain the state transition rules of CA models so that the models demonstrate the same behavior as the physical systems [12, 16, 23–27]. This ability to force correspondence between CA model behavior and the physical system increases the reliability of the CA models at least to the level of the macroscopic understanding of the underlying system. This suggests that the reliability of results obtained from CA models constructed with these techniques can be as good or better than any other computer models, even ion channel models.

1.5.3 Understanding

One of the greatest potential advantages of a computer model is that the model architecture can be chosen to provide detailed information about those aspects of system behavior which are most important to a particular study. This study was originally undertaken to test a particular anti-arrhythmia pacing technique. A previous study of a similar technique in an animal model had failed to provide sufficient information to test the technique since the technique did not work as expected and the cause of the failure could not be determined from the experiment [17]. Thus the animal study failed the requirement of providing sufficient *understanding* of the system.

To understand the usefulness of a computer model in this case, it is useful to start by considering these shortcomings of animal models. While animals provide the most extensive representation of physiologic detail, they are often of very limited value in

exploring mechanisms. This is because the physiologic state of the animal is relatively inaccessible.

A very sophisticated animal experiment may have mapping data, for example the transmembrane potentials, from all points on the epicardial surface[6]. Although such an experiment would provide an impressive amount of detail, even the entire epicardial surface represents only a two-dimensional slice of the electrical activity of the three-dimensional heart, and thus provides very little of the information that would be needed to determine the complete physiologic "state" of the experiment.

Without the ability to determine the experimental state of the animal heart at any time, and without the ability ever to exert fine control over that state, it is extremely difficult to determine the detailed effect of an intervention. While the overall effect will be clear, i.e. "did it work?", the details which would help determine why it worked or how it failed are difficult to determine.

The situation is quite similar in an ion channel model. In this case, the details *are* all accessible, since the state of the computer program can be examined at any time, but there is so much state information that it is difficult identify the quantities relevant to the experiment. One of the benefits of successful modeling is an understanding of the basic processes that govern the behavior of a system. Such an understanding can be used to make predictions of weaknesses in the system (e.g. the effects of certain types of disease) or to aid the development of successful interventions. A model incorporating the level of detail necessary for an ion channel model is of very limited use beyond the microscopic realm because there is so *much* information that it is difficult to extract any fundamental relationships between the macroscopic parameters that control the behavior of the system. This severely limits the usefulness of these models in understanding macroscopic phenomena such as arrhythmia.

There is a similar problem in manipulating the state of the model, a useful ex-

[6]See, for example, [30], which represents the state of the art in such experiments.

perimental technique for isolating the causes of behavior. In a model which depends on a very large number of coupled, empirical parameters, any attempt to modify a property of the model will necessarily effect all other properties in the model through the coupling of parameters. This severely limits the ability to do certain types of numerical "experiments" on the models. Therefore, a (disease) process can often not be modeled until *after* the researcher has a detailed understanding of the ionic basis for that process.

Net current models are much more accessible than ion channel models and much easier to interpret, since they are designed around wave behavior. They do not, therefore, have the problem of providing too much information to interpret. They do suffer from the second problem: difficulty in manipulating the model state or model behavior. This is because the model response is abstracted into a small number partial differential equations which give the observed wave behavior of the system. The parameters in these equations are tightly coupled and therefore an attempt to modify one aspect of model behavior, e.g. to account for an empirical observation of a disease process, would unintentionally modify other aspects of model behavior.

It is in the requirement of understanding that CA models seem superior to all other types of models. Like net current models, CA models are designed to reproduce the macroscopic wave behavior that is critical to the understanding of arrhythmogenesis. CA models do not introduce the tight coupling of parameters characteristic of ion channel models or of net current models. Rather, as I will demonstrate in chapters 2 and 3, each important independent aspect of the behavior of the system being modeled can be separately represented in the state transition rules for a consistent CA model. This allows manipulation of the detailed behavior of the model to correspond to empirically-determined models of pathology and arrhythmogenesis. It also allows complete specification of model behavior in terms of high-level, macroscopic tissue parameters. The resulting model behavior, and the resulting *understanding*

of model behavior, is thus closely tied to intuitive, empirically-observed aspects of actual myocardial behavior.

One of the main justifications for the use of CA models has been that they have long been the only models that could practically be used to model arrhythmogenesis on computers. With the constant advance in computer technology CA models will soon lose their solitary status with respect to practicality. Yet CA models will likely continue to be important for the foreseeable future because they will continue to provide the best system for the abstraction and understanding of the *essential* aspects determining system behavior. This is the most important reason I have chosen a CA model for this study, and for this reason it is likely that CA models will continue be most important for computer studies of arrhythmia.

1.6 Thesis Layout

The remainder of this thesis will describe my construction of a finite-state cellular automata model of the ventricles, and my use of that model for exploring certain aspects of ventricular arrhythmogenesis and novel pacing protocols. In chapter 2 I will described in detail all the mathematical constraints placed on the behavior of the model to ensure that it does not violate the physical properties of the system being modeled. In chapter 3, I will then describe the detailed implementation of a model of the human heart, including the model used to introduce pathology.

In chapter 4, I will use the simulator to study specific local mechanisms which lead to the development of lethal arrhythmia. In chapter 5, I will develop a model of electrical cardiac disease and use a technique similar to provocative electrophysiologic testing to induce reentrant arrhythmias. I will then test diffuse-field pacing by modeling the effect of a completely homogeneous pacing field on the ventricle. I will try to stabilize the simulated myocardial substrates against arrhythmia by pre-

emptively pacing into provocative events *before* the arrhythmia has developed. From this, I hope to determine why the previous study failed to prevent arrhythmia, and to determine whether the technique can be improved.

Chapter 2

Mathematical Methods

The propagation of depolarization wavefronts in myocardial tissue is governed by the interaction between transmembrane ion concentrations and voltage-dependent membrane channels which control the flow of those ions. Because the channels are spatially distributed along the myocardial membranes, and because the ions diffuse through the intracellular and extracellular space, distant regions of tissue are diffusively coupled.

The state of the ion channels, and therefore the evolution of the transmembrane ionic concentrations, depends in general on the transmembrane voltage. That voltage, in turn, depends on the ionic concentrations, so the resulting system must be represented by sets of coupled differential equations for each ion species. Such a formulation was first developed for nerve tissue by Hodgkin and Huxley [38], and has since been extended to cardiac tissue.

It might seem that any attempts to model the evolution of depolarization wavefronts in the heart would ideally incorporate all the interaction between the ionic environment and the channels. There has been extensive work on such simulations [11, 46–48], which have been especially useful for making quantitative determinations of the dynamic behavior of the cardiac action potential under a wide variety of conditions (see, for example [20]).

As discussed in Section 1.5, this approach to computer modeling has several significant disadvantages. The first problem is obviously the complexity of such a model, limiting studies to two-dimensional regions of tissue on the order of a few cm^2. The second problem is that ion channel models are based on an empirical understanding of the ion channels themselves, rather than on large-scale behavior. The most important problem with ion channel simulations from our standpoint, however, problem is their enormous intricacy. A model incorporating the level of detail necessary for an ion channel model is of very limited use beyond the microscopic realm because there is so *much* information provided that it is difficult to extract any fundamental relationships between the macroscopic parameters that control the behavior of the system. This severely limits the usefulness of these models in understanding macroscopic phenomena such as arrhythmia.

Implementation of a computer model of cardiac conduction therefore first requires the determination of how much microscopic detail must be represented at the cost of efficiency and clarity in representing macroscopic phenomena. Though it is not possible to determine *a priori*, which microscopic details must be included in a model to give a faithful macroscopic representation, there are certain requirements which are necessary, if not sufficient, for faithful representation of phenomena. Such requirements can generally be classified as either reproduction of observed physiologic behavior, such known duration of action potential under varying circumstances; or reproduction of derived laws of physics based on the physical principles underlying the phenomena (such as the conservation of charge).

In this chapter, I will describe those mathematical constraints which can be placed on any system which seeks to reproduce observed myocardial behavior. I will demonstrate the basis of the choice of each of the principles governing simulator operation, and the validation of each of these choices against physical principles. Many of the principles presented here for developing correspondence between CA models and con-

tinuous media have also been presented in [12,16,23–27]. They will represent a series of constraints on simulator behavior. A final determination of the significance of results from the simulator will, of course, still depend on the range of phenomena observed.

In section 2.1, I will provide a simple derivation of the one-dimensional cable equation which is the basis for understanding the behavior of any distributed, diffusively-coupled system, especially systems possessing excitability such as myocardial tissue. I will show what types of wave behavior these systems can support. In section 2.2, I will then give a general description of the implementation of any finite-state automata model, giving the rules (and vocabulary) which determine model evolution.

In section 2.3, I will demonstrate that there are strict requirements on the way any individual automaton (element) interacts with its neighbors to ensure that the symmetry of the physical system those rules to avoid breaking the being modeled is not broken by the symmetry of the lattice geometry on which the model is constructed. In section 2.4, I will show how the rules are further constrained to give specified plane wave speeds.

From this, in section 2.5, I will demonstrate that the mathematical relationship between the propagation speed of a plane wave, the propagation speed of a wave of small curvature, and the diffusion constant in the medium uniquely specifies the relationship between the simulator time step and the number of elements with which any element interacts during a single time step. In section 2.6, I will then demonstrate that it is possible to transform these rules to provide anisotropic propagation. Finally, in section 2.7, I will show how to incorporate details of the recovery process into the state transition rules for the automata, and how this constrains the choice of simulator time step.

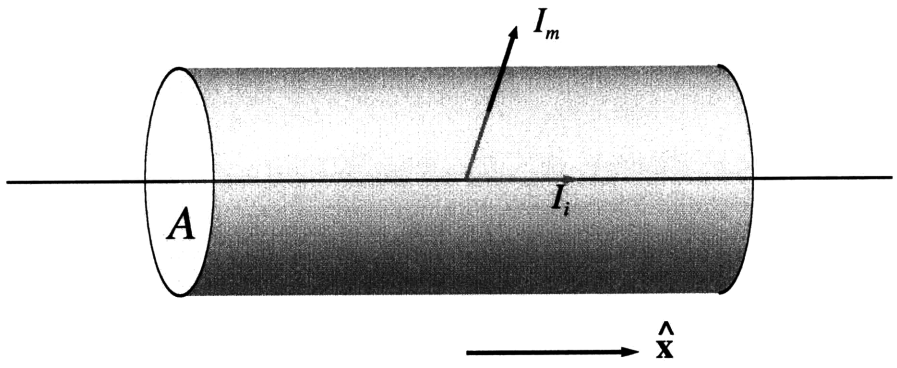
2.1 Reaction-Diffusion Systems

Myocardial tissue is a *reaction-diffusion* system. As the name implies, the behavior of such systems is controlled by two processes. *Diffusion* provides spatial coupling between the states of distant regions of the system. Locally, the process of diffusion can trigger a *reaction*: a rapid change in the local state of the system. A purely diffusive system cannot support traveling waves, since diffusion alone leads to a uniform steady state of the system. However, a spatially-distributed reaction-diffusion system such as myocardial tissue, is a type of *excitable medium*, which supports the propagation of solitary waves unattenuated over long distances [16].

2.1.1 One-Dimensional Cable Equation

The basis for understanding the physical properties of excitable media is the one-dimensional cable equation, which we will now derive. This derivation is based in part on the derivation of Keener [44], and leads to the basic equation describing excitable media, the Nagumo equation [54].

Figure 2-1 A One-dimensional "cable" of excitable media



Consider the system shown in Figure 2-1. There is a cylindrical membrane with cross-sectional area, A, and perimeter, p, separating and "inside" and an "outside"

region, each having specific ionic concentrations. We can assume that these concentrations vary in general with x (the axial direction), but not with r (the radial direction) except between the inside and the outside. This is the "core-conductor" assumption, which is good for long cables. Current can flow passively parallel to the horizontal axis either inside, with a resistivity R_i, or outside, with a resistivity R_o. Current can also flow "actively" through the membrane according to the membrane voltage given by the function $I_m = f(V,t)$, where I_m is the transmembrane current density and $f(V,t)$ can be any well-behaved function[1]. The membrane has an capacitance, C_m per unit area.

For simplicity we will take the outside (extracellular) resistance to be much less then the inside (intracellular) resistance, $R_o \ll R_i$. Although this is a simplification, it forms a good starting point for myocardial tissue [44]. This is equivalent to assuming the outside space is an equipotential, which we will take to be $V_o = 0$. The system is "one-dimensional" in the sense that it has cylindrical symmetry along the horizontal axis. Current flow inside the membrane is given by Ohm's law:

$$I_i(x) = \frac{A}{R_i}\frac{\partial V(x,t)}{\partial x} \tag{2.1}$$

Conservation of charge at any point, x, along the inside gives:

$$\frac{\partial I_i(x)}{\partial x} = pI_m, \tag{2.2}$$

where the membrane current is the sum of the transmembrane ionic current, given by $f(V,t)$, and the membrane capacitive current:

$$I_m = f(V,t) + C_m\frac{\partial V}{\partial t}. \tag{2.3}$$

Combining the above equations,

$$\frac{\partial}{\partial x}\left(\frac{A}{R_i}\frac{\partial V}{\partial x}\right) = p\left[C_m\frac{\partial V}{\partial t} + f(V,t)\right]. \tag{2.4}$$

[1]The term "active" as used here means only that $f(V,t)$ can be any $i-v$ relationship and is not constrained to conserve energy.

The gives the voltage in x, and t as a second-order partial differential equation. After rearrangement, it is easy to recognize the diffusion equation for a uniform cable with a source term:

$$\frac{\partial V}{\partial t} = \frac{A}{R_i p C_m} \frac{\partial^2 V}{\partial x^2} - \frac{f(V,t)}{C_m}, \tag{2.5}$$

where we identify the diffusion constant, D:

$$D \equiv \frac{A}{pC_m R_i},$$

where D has units of length^2/time and for myocardial tissue is typically $\approx 1\,\mathrm{cm}^2/\mathrm{sec}$.

2.1.2 Wave Behavior

The usual wave equation in physics is a second-order partial differential equation in space and time of the form (in one dimension):

$$\frac{\partial^2 v}{\partial x^2} + \frac{1}{c^2}\frac{\partial^2 v}{\partial t^2} = 0. \tag{2.6}$$

This is a hyperbolic partial differential equation and clearly supports wave solutions because the basis for the solution set is the complex exponentials, $e^{i(x-ct+\phi_i)}$, where there will be two distinct ϕ_i to satisfy the boundary conditions of the problem.

The diffusion equation, (2.5) differs from the wave equation (2.6) in that the time derivative is only first-order, meaning that this equation does not, in general, support wave solutions. Specifically, the system described by (2.5) is dissipative and can only support propagating solutions if $f(V,t)$ can "reinject" the energy dissipated in the passive term of the equation, i.e. if the $i-v$ relationship possesses *excitability.*

Starting with this requirement on the behavior of $f(V,t)$, we will consider two classes of wave solutions to (2.5): trigger waves and solitary waves. To do this, we start by assuming the existence of traveling waves in the physical system, and thus the existence of a stationary (time-invariant) traveling wave solution at speed, c, for

the particular $f(V)$ which describes the system. To find these solutions, we first transform to a moving coordinate system with x and t combined in a single "phase" coordinate, ξ, and require that wave solution be invariant:

$$\xi = x - ct \qquad t' = t, \tag{2.7}$$

where we have transformed $t' = t$ to preserve the dimensionality of the problem (i.e. leave the transformation invertible).

Trigger waves occur in systems with two distinct stable states, which we will call the "resting" and "excited" states, labeled as V_- and V_+. Such systems are called *bistable media.* The trigger wave is a propagating transition (a "moving boundary layer" [16]) between the two stable states of the system[2]. For a wave traveling in the positive-x direction, we require the boundary conditions:

$$\begin{aligned} \lim_{\xi \to -\infty} u(\xi) &= V_- \\ \lim_{\xi \to \infty} u(\xi) &= V_+. \end{aligned} \tag{2.8}$$

The existence of these two stable states requires that $f(V)$ have at least two stable roots (positive curvature), and therefore at least one intermediate unstable root (negative curvature), which corresponds to the threshold voltage for the system, V_{th}. This clearly requires that $f(V)$ be a non-linear, non-monotonic function. This is an important distinction between (2.6) and (2.5): wave behavior in (2.5) can *only* be due to non-linearities in $f(V)$ and can therefore only be nonlinear processes. More significantly, the existence of three real roots requires that $f(V)$, the $i - v$ relationship for the problem, change sign at least three times. It is the portions of "negative resistance" in this $i - v$ relationship which provide the excitability in the system and allow wave solutions to a dissipative system.

[2] A very simple example of such a system is a row of dominoes falling.

Transforming to the new coordinates yields:

$$\frac{\partial}{\partial x} = \frac{\partial}{\partial \xi} \tag{2.9}$$

$$\frac{\partial}{\partial t} = c\frac{\partial}{\partial \xi} + \frac{\partial}{\partial t'}. \tag{2.10}$$

In these variables, (2.5) becomes,

$$c\frac{\partial u}{\partial \xi} + \frac{\partial u}{\partial t'} = D\frac{\partial^2 u}{\partial \xi^2} + f(u).$$

A stationary "traveling-wave" solution is one in which the wavefront does not change when observed from the traveling frame: $\partial u/\partial t' = 0$, therefore,

$$D\frac{\partial^2 u}{\partial \xi^2} - c\frac{\partial u}{\partial \xi} + f(u) = 0. \tag{2.11}$$

Equation (2.11) is simply a rearrangement of (2.5) and does not therefore *prove* the existence of traveling wave solutions to the problem. It is simply our hopeful statement that there *are* traveling solutions to the problem. Although it is theoretically possible to prove the existence of wave solutions for certain special classes of $f(V)$ [16,44], that is not necessary in this case. The system is experimentally known to support wave solutions with a particular plane wave speed, c, and we have set up the equation for the purpose of studying those solutions. Equation (2.11) is therefore important because it gives the form of the equation which must be satisfied by any observed stationary traveling wave solution of the system under study, and therefore constrains the properties that such a solution can have. This will be discussed in detail in section 2.5.

Trigger waves are "one-shot" phenomena, having no recovery process, and are therefore simpler than the traveling waves that occur in excitable media like heart tissue. *Solitary waves* occur in systems with a stable state (resting) and a metastable state (excited). There is then a traveling wavefront—the transition between resting and excited—and a traveling waveback—the transition back from excited to resting[3].

[3]For a description of another physical system which undergoes a rapid transition between two metastable states, see [19]

The description of the metastable excited state requires the addition of a time dependence to $f(V)$, usually through the introduction of a "recovery" variable, u:

$$\frac{\partial V}{\partial t} = D\frac{\partial^2 V}{\partial x^2} - f(V,u) \tag{2.12}$$

$$\frac{\partial u}{\partial t} = g(V,u). \tag{2.13}$$

In myocardial tissue, the recovery process, $g(V,u)$ most closely approximates the slow repolarizing currents.

Again, finding wave solutions to a set of coupled partial differential equations of this type can only be done for certain special classes of f and g. The knowledge gained from the trigger-wave limit can be exploited for analyzing this system, however, because of a fortuitous coincidence: the transition time for (2.12) i.e., the period during which $\partial V/\partial t$ is large, is very short compared to the transition time for (2.13). We define τ_V and τ_u as the characteristic times for (2.12) and (2.13), respectively. We then rescale the time and space dimensions in the problem to incorporate these characteristic scales:

$$t \rightarrow \frac{t}{\tau_V} \tag{2.14}$$

$$x \rightarrow \frac{x}{\sqrt{D\tau_V}}, \tag{2.15}$$

we can rewrite (2.12) and (2.13) as [16]:

$$\frac{\partial V}{\partial t} = \frac{\partial^2 V}{\partial x^2} - f(V,u) \tag{2.16}$$

$$\frac{\partial u}{\partial t} = \epsilon g(V,u), \tag{2.17}$$

where $\epsilon \equiv \tau_V/\tau_u$. Since $\epsilon \ll 1$, recovery as described by (2.17) can now be treated as a perturbation on the trigger wave problem. This means the methods for trigger waves can be used to determine the first-order behavior of the wavefront and to describe the waveback. We will use this fact to decouple the study of excitation from the recovery, later adding recovery as a perturbation.

2.2 Finite-State Automata

A finite-state cellular automaton (CA) is an element, usually part of collection of many similar elements, which is always in one of a specified number of internal *states* and who's internal state evolves in time according to a specified set of *state transition rules.* Such an abstraction is very powerful for modeling, especially on computers, certain types of systems.

Before describing the details of finite-state cellular automata, it is interesting to examine the flexibility of the system. In fact, any model that can be implemented on a digital computer can be implemented as a CA model. As an extreme example, consider an implementation of a Hodgkin-Huxley-type model of cardiac tissue. Such a computer model could keep track of the concentrations at each point of Na^+, K^+, and Ca^{++}, from which a transmembrane voltage would be calculated. This would determine the state of each ion channel, which would then give the coefficients in the partial-differential equation determining the change in the ionic concentrations during the next simulator time step.

To implement this system as a finite-state CA, one associates with each discrete lattice point a single CA (or element). The *state* of that element is the binary representation of each of the ionic concentrations (since it is a binary representation on a computer, it can only take on certain discrete values and is therefore part of a finite set). This is said to be the "limit-continuous" case, because the states are used to approximate continuous behavior. The *state transition rules* mapping each state to the next are given by the channel kinetics and the differential equations. This is obviously an extreme example, but it is sometimes easy to assume that the relatively primitive models usually implemented as finite-state cellular automata are a reflection of limitations to the CA approach. In fact, a well-designed CA model can implement models of arbitrary sophistication.

2.2.1 Minimal States and Transition Rules

Since myocardial tissue is an *excitable medium*, any finite-state model must necessarily have at least two states: EXCITED and RESTING. Within the EXCITED state, it is also useful to distinguish between times in which the element can affect the state of its neighbors, i.e. when it is EXCITING; and those when it cannot, i.e. when it is REFRACTORY.

For this chapter, we will work with a minimum set of states and state transitions rules for an excitable medium. Myocardial tissue will require a more complex system, but the minimal set here is sufficient to determine all mathematical constraints on the transition rules developed in Chapter 3 for myocardium.

The following is a minimum set of state transition rules sufficient for modeling a spatially-extended excitable system:

1. The state of each element can be externally effected only by elements in the *neighborhood* of the element in question. The *neighborhood* of an element is some region local to the element. In this model, any operation performed on a neighborhood will treat all elements within the neighborhood as identical. This means, for example, the distance of a neighborhood element to the central element is not considered.

2. Only RESTING elements may have their states effected externally. EXCITED elements (EXCITING and REFRACTORY) elements are insensitive to external influence[4].

3. Only EXCITING elements are capable of affecting the states of other elements.

4. A RESTING element will become EXCITED in the next simulator time step if and only if the number of EXCITING elements in its neighborhood exceeds its

[4]This represents a simplification for the general behavior described by (2.17), but is quite good for myocardial tissue since recovery is not strongly coupled in space.

excitation threshold.

5. An EXCITING element will become REFRACTORY (no longer able to excite other elements) at a predetermined time after excitation.

6. A REFRACTORY element will become RESTING at a (longer) predetermined time after excitation.

Understanding these three minimal states and minimal state transition rules is sufficient for understanding the constraints described in this chapter. In general, the states and rules will be more complex, but will always represent a direct generalization of this minimal set.

2.3 Modeling Systems with Continuous Symmetry

One of the most useful principles of physics is symmetry. Understanding of symmetry provides a vast simplification of physical phenomena because *it is not generally possible for observed phenomena to demonstrate a different symmetry than the physical interactions that produce those phenomena.* This is the basis for all conservation laws. The symmetry of the interaction therefore places significant constraints of any phenomena resulting from that interaction.

Symmetry is equally important in understanding wave phenomena in myocardial tissue. The problem is that the relevant symmetry is much more difficult to identify. At the microscopic level there is no symmetry to myocardial tissue. It is a collection of irregularly-shaped cells, connected by irregularly-spaced gap junctions, with excitation controlled by irregularly-spaced ion channels. Though this might appear to be a hopeless situation, a completely irregular microscopic system provides perfect macroscopic symmetry because in a completely irregular microscopic system there is no *preferred* direction. This suggests that the starting point for modeling

macroscopic phenomena (wave propagation), should be circular symmetry, which is the only system with no preferred direction.

This situation is not quite that simple of course, because myocardial tissue generally does have a preferred direction: the direction of local fiber orientation. This results in the observed differences in propagation speed along the longitudinal compared to the transverse axis [69, 72]. This type of symmetry is however only a small modification of continuous rotational (circular) symmetry (Sec. 2.6). We will start, therefore, with a consideration of how the system would be modeled if there were truly no preferred direction (random alignment of myocytes), and then modify the model to account for the preferential alignment of myocytes along one axis.

2.3.1 Symmetry of Discrete Lattices

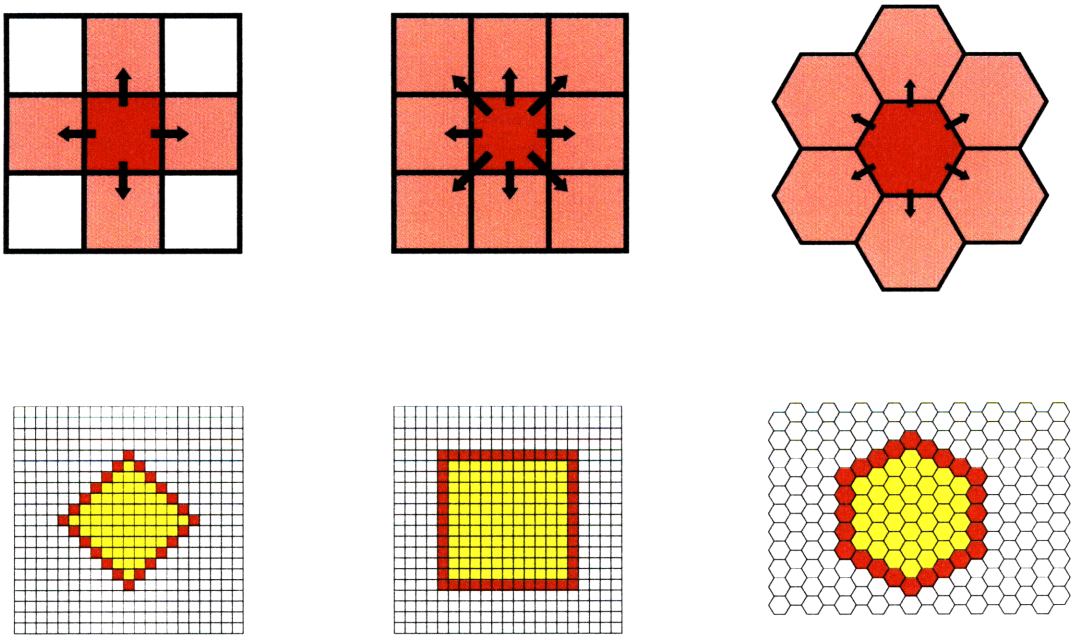
The construction of a finite-state cellular automata model requires the discretization of the system being modeled into a finite set of elements. In a two-dimensional system, these elements will fit together to occupy the entire area of the system. Usually, all elements are taken to be the same size and shape[5]. It is not difficult to show that there are only five such lattices in two dimensions, and each has discrete symmetry, i.e., a small number of *preferred directions.* The two most commonly used lattices for two-dimensional finite element simulations have lattices square or hexagonal elements (square and triangular lattices, respectively). The square lattice is probably most often used because the locations of the centers of each lattice element form an orthogonal coordinate system.

A difficulty with square lattices is the fact that there are two distinct sets of "nearest" neighbors (Figure 2-2(a), (b)). The distance from a lattice point to a "nearest" neighbor in the horizontal or vertical direction (an "edge" neighbor) is the lattice spacing. The distance to a "nearest" neighbor in the diagonal direction

[5]Such lattices are called *Bravais Lattices*

Figure 2-2 Two simple Bravais lattices (top) and the waves resulting from nearest-neighbor interactions on those lattices (bottom). Figures (a) and (b) show square lattices (each lattice point is at the center of an identical square). In Figure (a), the "nearest-neighbors" are taken to be those lattice elements that share an edge with the central element. In Figure (b), "nearest-neighbors" are taken to be any element in contact with the central element. Each system produces square waves from a single element, though the wave are oriented at 45° to each other. Figure (c) shows a hexagonal lattice. Although this eliminates the problem of choosing between two different types of neighbors, the resulting wave still has the (discrete) symmetry of the lattice. From [57].

(a) (b) (c)



(a "corner" neighbor) is $\sqrt{2}$ times the lattice spacing. This creates an unsolvable paradox for establishing nearest-neighbor propagation rules. If the nearest neighbors (those neighbors which will excite one time step after the central element) are taken to be the "edge" neighbors (Figure 2-2(a)), then the "corner" neighbors will excite *two* time steps later, despite being only *root two* further distant. The propagation speed is therefore different in orthogonal directions than at 45°. The situation is inverted, but not corrected, by using the propagation scheme shown in Figure 2-2(b).

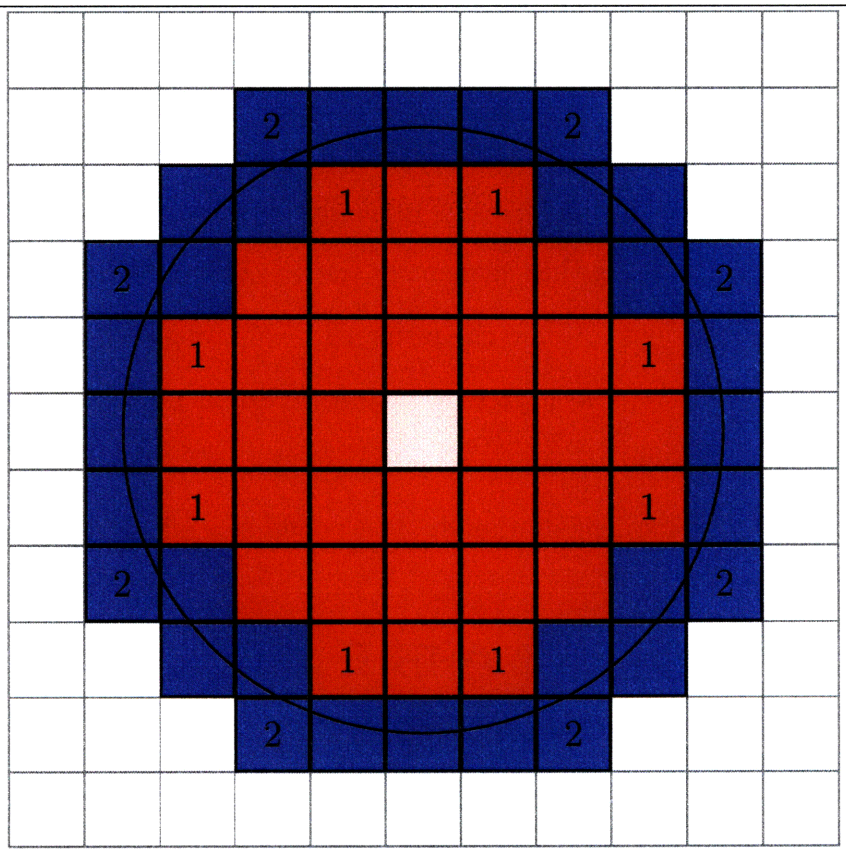
The problem of unequal directions to lattice elements is apparently addressed by *hexagonal* elements. These elements are on an equilateral triangular lattice, having six equivalent nearest neighbors (Figure 2-2(c). This eliminates the paradox presented by the square lattice, since all adjacent neighbors are separated by the lattice spacing. The problem is deferred only for one time step, however. Since the (six-fold) interaction rule has a hexagonal symmetry, the resulting waves can only be hexagonal. This is seen in the bottom panel of Figure 2-2(c). Those elements at the vertices of the dark hexagon excite simultaneously with the elements in the center of the hexagonal faces, despite being further away by a ratio of $2{:}\sqrt{3}$. The preferred (fast) propagation direction is therefore toward elements aligned with the vertices of the lattice elements, compared to those aligned with the edges, and the symmetry is discrete.

There have been some attempts to impose nearly-continuous symmetry on a discrete lattice by using a collection of elements of different sizes and shapes [55], but such lattice geometries only succeed in creating higher-order polynomials–the problem of preferred propagation directions continues to exist [50].

2.3.2 Continuous Symmetry on Discrete Lattices

A solution to the problem of modeling continuous symmetry on discrete lattices is shown in Figure 2-3. Consider first the neighborhood represented by the elements in red. This is an extended interaction neighborhood which is defined in this case

Figure 2-3 Determination of Neighborhood for Circular Symmetry. Red elements are *always* included in neighborhood. This alone would give polygonal symmetry, with preferred propagation directions towards elements marked with 1. Blue elements are *statistically* included in the neighborhood to give circular symmetry *on average*. Thus, those elements which would represent preferred propagation directions (those marked 2) are *least likely* to be in the neighborhood.



by those elements that are *completely within* a specified circle. The neighborhood of the central element therefore includes elements that are several lattice steps away, and all elements in the neighborhood are considered equally with respect to the central element (that is, there is no distinction within the neighborhood based on distance to the central element). It is clear that such a neighborhood is visually a better approximation to a circular (continuous symmetry) neighborhood, but in fact it is only another discrete approximation, as is the neighborhood consisting of all elements which are not completely outside the circle (red and blue elements). These neighborhoods still have preferred (rapid propagation) directions toward elements whose outer corners are radially most distant from the central element.

Markus and Hess [49] realized that the problem of *preferred propagation directions* in a discrete lattice could be eliminated by randomly choosing a fiducial point for each lattice element, and including that element in the neighborhood based on the radial distance to its fiducial point. While the resulting neighborhood would be polygonal (since it is the union of square lattice elements), it would be impossible to predict which outer elements would be in the neighborhood and there could therefore be no preferred directions to the neighborhood. The neighborhood would be isotropic *on average.*

In the model I have adapted this technique by creating an *ensemble* of neighborhoods with a given interaction radius, R, constructed according to the following rules:

1. Any element entirely within the circle of radius R (red elements) will be included in each neighborhood in the ensemble.

2. Any element entirely outside the circle of radius R (white elements) will not be included in any neighborhood in the ensemble.

3. Any element partially within the circle of radius R (blue elements), will be

included in a neighborhood of the ensemble with a *probability* given by the fraction of the element which is inside the circle.

For each element, at each excitation, a neighborhood is randomly chosen from this ensemble. While each such neighborhood is polygonal, the polygons vary both in space and time, and it is therefore impossible to predict and preferred direction. On average, spatially and temporally, the interaction neighborhood of any element is circular, and phenomena produced by the simulator are therefore continuously symmetric, as required by the underlying interactions. The method of randomly choosing neighborhoods from an ensemble allows the incorporation of temporal averaging, in addition to the spatial averaging given by Markus and Hess [49]. It is straightforward to modify this system for the anisotropy observed in myocardial tissue, as will be discussed in section 2.6.

2.4 Modeling Propagation of Plane Waves

To establish quantitative behavior of the model, it is necessary to determine exactly the conditions for excitation as a function of the state of elements in the neighborhood. In section 2.3.2, I described a technique for providing and effectively-circular interaction neighborhood. I will now show how that neighborhood is used to provide a continuous range of plane-wave propagation speeds.

Consider an element with a circular interaction neighborhood of radius R (the determination of R is left to section 2.5). Since all elements in the interaction neighborhood are treated equivalently, excitation of the element can only depend on the *number* of elements in the neighborhood, or equivalently the *fraction* of the elements in the neighborhood which are in a state that will trigger excitation. Define the *excitation threshold* for the element at the center of the neighborhood, K, as that fraction of elements in the neighborhood which must be EXCITING to cause excitation of the

central element.

Figure 2-4 Geometric Relationship of Excitation Threshold to Plane Wave Speed. A plane wave (red) traveling at speed c is approaching the central element of the neighborhood shown (green). Since the plane wave is traveling at speed c, it will *just* excite the central element at the next time step. The threshold area will be the overlap of the wavefront and the neighborhood (yellow).

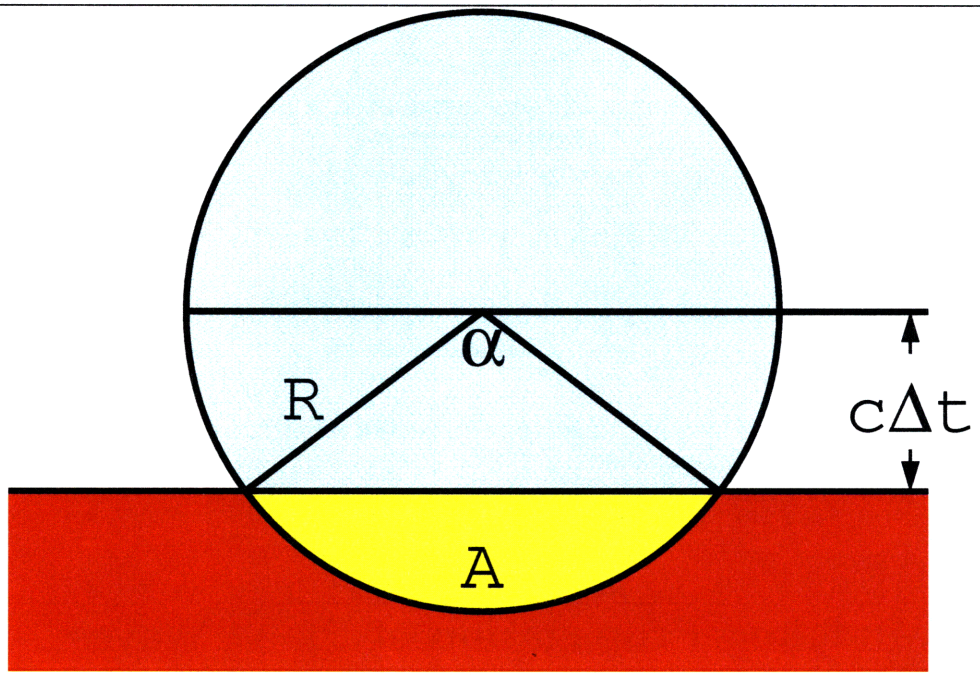


Figure 2-4 shows the relationship of an oncoming plane wave, consisting of EXCITING elements and a central RESTING element. The plane wave is shown in red and for this discussion we may assume is extends arbitrarily far back (the trigger wave limit). The neighborhood of the element under consideration is shown as a circle. The constraint from which the threshold K will be determined is that plane waves propagate at speed, c. For a simulator time step of Δt, this condition is equivalent to the requirement

Constraint 1 *Any excitable element with a normal distance, d, to a plane wave front*

$$0 < d \leq c\Delta t$$

will excite during the next time step.

We will consider the case in which the wave propagates exactly $c\Delta t$ to the central element in the neighborhood shown in figure 2-4. The overlap between the neighborhood and the oncoming plane wave is the yellow area A. The area of this region, divided by the area of the circle gives the fractional threshold, K.

It is easiest to define the threshold in terms of α, the angle subtended by the oncoming wave. From figure 2-4,

$$c\Delta t = R\cos\frac{\alpha}{2} \tag{2.18}$$

To calculate the threshold area, A, let A_α be the area of the slice subtended by α:

$$A_\alpha = R^2\alpha/2,$$

and $A_\triangle$ be the area enclosed by the triangle with the base at the front of the oncoming wave and apex at the center of the circle:

$$A_\triangle = R^2 \sin\frac{\alpha}{2}\cos\frac{\alpha}{2}.$$

We then have (using the half-angle relations for $\sin\alpha$):

$$A = A_\alpha - A_\triangle = \frac{R^2}{2}(\alpha - \sin\alpha)$$

$$K \equiv \frac{A}{\pi R^2} = \frac{\alpha - \sin\sigma}{2\pi}, \tag{2.19}$$

where from (2.18),

$$\alpha = 2\cos^{-1}\frac{c\Delta t}{R}.$$

2.5 Modeling Propagation of Curved Waves

In section 2.4 we saw that it was possible to determine the excitation threshold of a neighborhood of radius R, from the plane wave speed, c, and the simulator time step,

Δt, by a simple geometric construction. We now consider the problem of choosing the appropriate value of R, the neighborhood size.

The neighborhood of an element consists of all the elements which that element "samples" during one time step. Since coupling in this model is by diffusion, it is reasonable to expect that this "sampling distance" should be a diffusion length, that is, the approximate distance a signal would diffuse during a single simulator time step. From unit analysis, an initial crude approximation would be

$$R \sim \sqrt{D\Delta t}.$$

With this as motivation, we seek a quantitative relationship between the diffusion constant D, and the dynamics of the system.

In section 2.1.1 we saw that trigger waves in a diffusively coupled excitable medium can be described by a second-order partial differential equation of the form:

$$\frac{\partial V}{\partial t} = D\frac{\partial^2 V}{\partial x^2} + f(V). \tag{2.20}$$

Depending on $f(V)$, the system may support plane traveling waves, which would then be solutions to:

$$D\frac{\partial^2 V}{\partial \xi^2} - c_0\frac{\partial V}{\partial \xi} + f(V) = 0, \tag{2.11}$$

where ξ is the local "phase" variable for the traveling wave, and c_0 is the speed identified in the one-dimensional case for the propagation of plane waves (since plane waves are effectively one-dimensional). We denote the speed of a plane wave as c_0 to distinguish it from the general speed, c, for an arbitrary traveling wave.

Consider now the problem in two dimensions for waves of small, finite curvature. The first correction to plane waves is the assumption that the wavefront is part of a circle of radius r, where r is very large compared to other lengths in the problem. It is therefore natural to take advantage of the symmetry of the problem by using polar

coordinates with the origin at the center of curvature:

$$\nabla^2 \equiv \frac{\partial^2}{\partial r^2} + \frac{1}{r}\frac{\partial}{\partial r}. \tag{2.21}$$

(In general, ∇^2 also has components in θ, but the system is symmetric in θ, so those terms vanish.)

Substituting r for x and rearranging, (2.11) becomes:

$$D\frac{\partial^2 V}{\partial \xi^2} + \left[c_0 + \frac{D}{r}\right]\frac{\partial V}{\partial \xi} + f(V) = 0. \tag{2.22}$$

(2.22) is valid in this case because radius of curvature, r, is changes very slowly as the front evolves. Comparison of this solution to (2.20) suggests the transformation:

$$c_0 \longrightarrow c = c_0 + \frac{D}{r}. \tag{2.23}$$

Since both equations must be correct (in the limit of small curvature), (2.22) must be correct in that limit. This gives the desired relation between the dynamics of the system and diffusion constant, from which we can specify R:

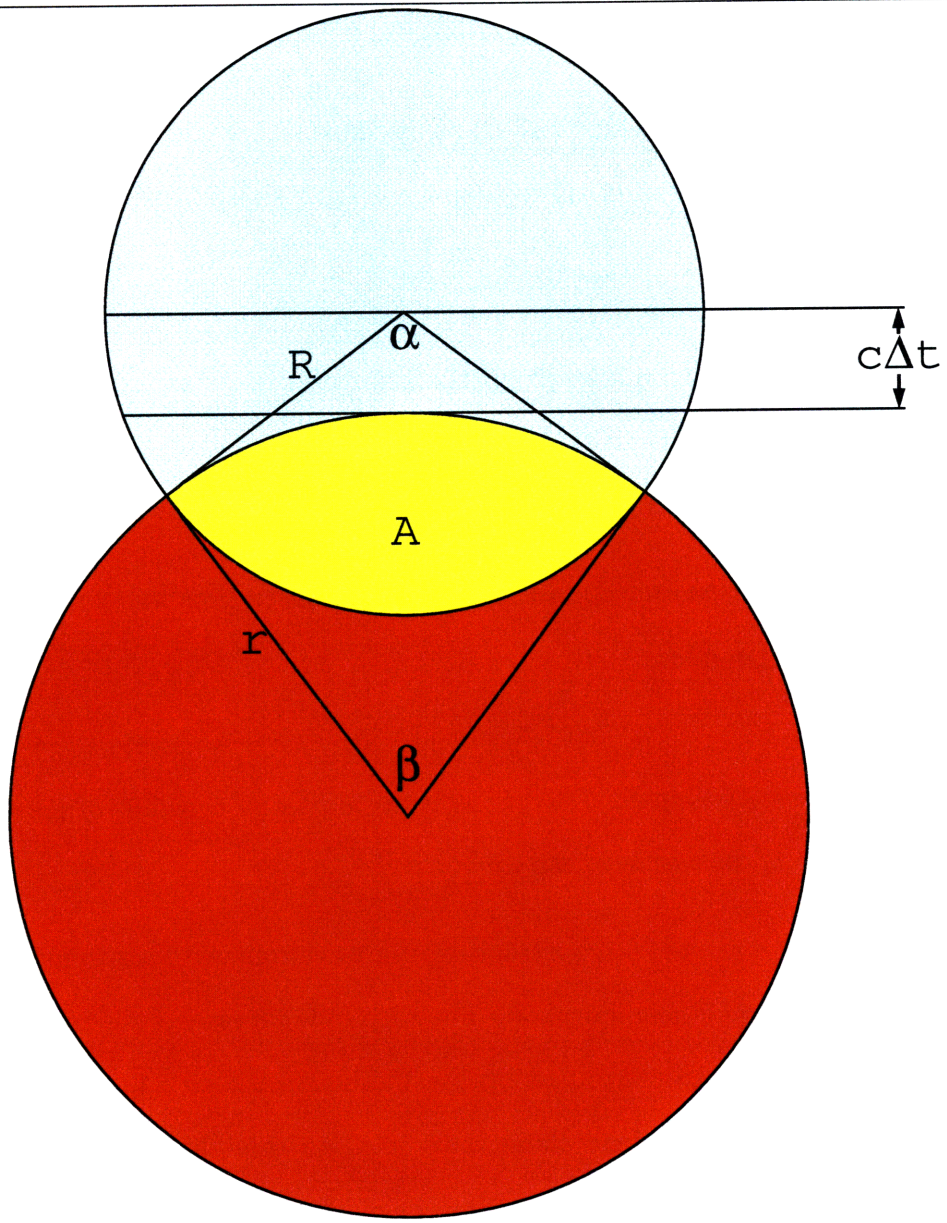
Constraint 2 *The neighborhood radius, R, must be chosen such that small curvature waves propagate with a speed*

$$c = c_0 + \frac{D}{r},$$

where c_0 is the plane wave speed, D is the effective diffusion constant of the medium, and r is the radius of curvature of the wave.

Figure 2-5 shows a geometric construction of a wavefront with a specific radius of curvature, r, incident on a neighborhood. The top circle represents the neighborhood whose threshold has been determined by the methods of Sec. 2.4. As before, the (normal) distance of the incident plane wave front to this element is taken to be exactly the distance to cause excitation in the next time step: the speed of the incident (curved) front is c. This is equivalent to the requirement that the area, A, is exactly the areal threshold of that element.

Figure 2-5 Geometric Relationship of Neighborhood Radius to Speed-Curvature Relationship. The curved wavefront is assumed to be propagating outward from the center of the lower circle (red), with a normal speed of c. It will therefore just excite the element at the center of the upper circle (green). The overlap between the (red) wave and the (green) element neighborhood is yellow. The relationship between this overlap area and the radius of curvature of the propagating wave, r, will determine the speed-curvature relationship.



As for plane waves, the task is to calculate $A(\alpha)$ given the other parameters, which in this case also include the radius of curvature of the incident front, r. Begin with a calculation of $c\Delta t$. The "position" of the wavefront is the point on the front closest to the center of the neighborhood. Comparing figure 2-5 to figure 2-4, the wave has already propagated further forward by a distance $r - r\cos\beta/2$. Therefore the distance it must travel during the next time step, $c\Delta t$ is decreased by that amount:

$$c\Delta t = R\cos\frac{\alpha}{2} - r\left(1 - \cos\frac{\beta}{2}\right). \tag{2.24}$$

Calculation of the area of the neighborhood covered by the incident curved wavefront can again be done by the methods of section 2.4, noting that the region shown in figure 2-4 does not overlap a similar region in the larger circle of figure 2-5, and thus the areas for both circles can be simply added:

$$A = \frac{R^2}{2}(\alpha - \sin\alpha) + \frac{r^2}{2}(\beta - \sin\beta). \tag{2.25}$$

We now have equations in this new geometry for both $c\Delta t$ and A. We eliminate β by noting that the angle in each of the circles describes a chord of the same length:

$$2R\sin\frac{\alpha}{2} = 2r\sin\frac{\beta}{2},$$

which allows us to solve for β in terms of the radius of curvature, r:

$$\begin{aligned}
\sin\frac{\beta}{2} &= \frac{R}{r}\sin\frac{\alpha}{2} \\
\cos\frac{\beta}{2} &= \sqrt{1 - \frac{R^2}{r^2}\sin^2\frac{\alpha}{2}} \\
\beta &= 2\sin^{-1}\left(\frac{R}{r}\sin^2\frac{\alpha}{2}\right)
\end{aligned} \tag{2.26}$$

Using this, we can reexpress (2.24) in r only:

$$c\Delta t = R\cos\frac{\alpha}{2} - r\left(1 - \sqrt{1 - \frac{R^2}{r^2}\sin^2\frac{\alpha}{2}}\right), \tag{2.27}$$

then expand the square root, since $R/r \ll 1$ (the small-curvature limit):

$$\begin{aligned} c\Delta t &\approx R\cos\frac{\alpha}{2} - r\left(1 - \left(1 - \frac{R^2}{2r^2}\sin^2\frac{\alpha}{2}\right)\right) \\ &= R\left(\cos\frac{\alpha}{2} - \frac{R}{2r}\sin^2\frac{\alpha}{2}\right). \end{aligned} \tag{2.28}$$

We then use (2.26) to solve for the perturbed A in terms of r:

$$A = \frac{R^2}{2}(\alpha - \sin\alpha) + r^2\left[\sin^{-1}\left(\frac{R}{r}\sin\frac{\alpha}{2}\right) - \frac{R}{r}\sin\frac{\alpha}{2}\sqrt{1 - \frac{R^2}{r^2}\sin^2\frac{\alpha}{2}}\right]. \tag{2.29}$$

This gives an analytic expression for the threshold area, A, for arbitrary small curvature. For the purposes of this section, we seek only the first-order correction in $1/r$ to the speed, c, so we can impose Constraint 2. We will first determine the first-order correction in curvature to α to give the angle subtended by the incident wave. We will then use the corrected α to determine the corrected speed, c, from which we will choose R according to Constraint 2.

Our approach will be to expand A to first-order in the two (dependent) parameters that have changed from the plane-wave case: the curvature, $\kappa \equiv 1/r$, and the angle subtended by the incident wave, α. Since the threshold of this element is given by the plane wave speed (Constraint 1) and is therefore independent of κ or α for the incident wave the dependence of A on the new parameters must cancel:

$$dA = \frac{\partial A}{\partial \kappa}d\kappa + \frac{\partial A}{\partial \alpha}d\alpha = 0. \tag{2.30}$$

We will then be able to rearrange (2.30) to calculate $d\alpha/d\kappa$, from which we will determine the correction to α. We will substitute that expression into (2.28) to determine the relationship between speed and curvature.

We first reexpress A in terms of α and κ:

$$A = \frac{R^2}{2}(\alpha - \sin\alpha) + \frac{\sin^{-1}\left(R\kappa\sin\frac{\alpha}{2}\right) - R\kappa\sin\frac{\alpha}{2}\left(1 - R^2\kappa^2\sin^2\frac{\alpha}{2}\right)^{1/2}}{\kappa^2}.$$

$\partial A/\partial\kappa$ can be found by direct differentiation, but it is most convenient to instead expand A in the small parameter $\epsilon \equiv R\kappa$:

$$A = \frac{R^2}{2}(\alpha - \sin\alpha) + r^2 \left[\underbrace{\sin^{-1}\left(\epsilon \sin\frac{\alpha}{2}\right)}_{\boxed{1}} - \epsilon \sin\frac{\alpha}{2} \underbrace{\left(1 - \epsilon^2 \sin^2\frac{\alpha}{2}\right)^{1/2}}_{\boxed{2}} \right]. \tag{2.31}$$

Taking $\epsilon \ll 1$, terms $\boxed{1}$ and $\boxed{2}$ in (2.31) can be expanded:

$$\boxed{1}: \qquad \epsilon \sin\frac{\alpha}{2} + \frac{1}{6}\left(\epsilon \sin\frac{\alpha}{2}\right)^3 + \ldots, \tag{2.32}$$

and

$$\boxed{2}: \qquad 1 - \frac{\epsilon^2}{2}\sin^2\frac{\alpha}{2}. \tag{2.33}$$

Substitution into (2.31) gives:

$$A \approx R^2 \left[\frac{1}{2}(\alpha - \sin\alpha) + \frac{1}{\epsilon^2}\left(\epsilon \sin\frac{\alpha}{2} + \frac{1}{6}\left(\epsilon \sin\frac{\alpha}{2}\right)^3 - \epsilon \sin\frac{\alpha}{2} + \frac{\epsilon^3}{2}\sin^3\frac{\alpha}{2} \right) \right] \tag{2.34}$$

Preserving only first-order terms in ϵ,

$$A \approx R^2 \left[\frac{1}{2}(\alpha - \sin\alpha) + \frac{2\epsilon}{3}\sin^3\frac{\alpha}{2} \right]. \tag{2.35}$$

Since $\epsilon \equiv R/r \equiv R\kappa$ and $1/r \ll 1/R$, (2.35) is equivalent to the statement:

$$\frac{\partial A}{\partial \kappa} = -\frac{2R^3}{3}\sin^3\frac{\alpha_0}{2}. \tag{2.36}$$

Direct differentiation of (2.35) with respect to α gives:

$$\frac{\partial A}{\partial \alpha} = R^2 \left[\frac{1}{2}(1 - \cos\alpha) + \epsilon \sin^2\frac{\alpha}{2}\cos\frac{\alpha}{2} \right]. \tag{2.37}$$

From (2.30), (2.36), and (2.37) we have:

$$\frac{d\alpha}{d\kappa} = \frac{4R^3}{3R^2}\frac{\sin^3\frac{\alpha_0}{2}}{(1 - \cos\alpha_0)}, \tag{2.38}$$

where we have ignored the term in (2.37) that is also first-order in ϵ. Using the half-angle formula for cosine, the denominator of (2.38) can be expanded:

$$\frac{d\alpha}{d\kappa} = -\frac{4R}{3}\left[\frac{\sin^3\frac{\alpha_0}{2}}{2(1-\cos^2\frac{\alpha_0}{2})}\right] = -\frac{2R}{3}\sin\frac{\alpha_0}{2}, \tag{2.39}$$

from which,

$$d\alpha = \frac{d\alpha}{d\kappa}d\kappa = \frac{d\alpha}{d\kappa}\left(\frac{1}{r}\right) = -\frac{2}{3}\epsilon\sin\frac{\alpha_0}{2}. \tag{2.40}$$

We now substitute $\alpha = \alpha_0 + d\alpha = \alpha_0 + (d\alpha/d\kappa)d\kappa$:

$$\alpha = \alpha_0 - \frac{2}{3}\epsilon\sin\frac{\alpha_0}{2} \tag{2.41}$$

into (2.28), expanding to first-order the terms in $d\alpha$, to get the corrected curved-wave speed, c:

$$c = \frac{R}{\Delta t}\left[\cos\frac{\alpha_0}{2}\cos\frac{d\alpha}{2} - \sin\frac{\alpha_0}{2}\sin\frac{d\alpha}{2} - \frac{\epsilon}{2}\left(\sin^2\frac{\alpha_0}{2} + d\alpha\,\sin\alpha_0\right)\right].$$

Dropping the cross term in $\epsilon\,d\alpha$, we can expand both terms involving $d\alpha$ to first-order:

$$\begin{aligned} c &= \frac{R}{\Delta t}\left[\cos\frac{\alpha_0}{2}\cos\frac{d\alpha}{2} - \sin\frac{\alpha_0}{2}\sin\frac{d\alpha}{2} - \frac{R}{2r}\sin^2\frac{\alpha_0}{2}\right] \\ &= \frac{R}{\Delta t}\left[\cos\frac{\alpha_0}{2} + \kappa\frac{R}{3}\sin^2\frac{\alpha_0}{2} - \kappa\frac{R}{2}\sin^2\frac{\alpha_0}{2}\right] \\ c &= c_0 - \kappa\left(\frac{R^2}{6\Delta t}\sin^2\frac{\alpha_0}{2}\right). \end{aligned} \tag{2.42}$$

(2.43) and Constraint 2 give the relationship between the curvature of the wave-front, κ, the effective diffusion constant of the medium, D, and the simulator time step, Δt, where the term in parenthesis in (2.43) is the effective diffusion constant, D:

$$R = \sqrt{\frac{6D\Delta t}{\sin^2\frac{\alpha_0}{2}}}. \tag{2.43}$$

2.6 Modeling Anisotropic Wave Propagation

All the relations derived so far have been for completely isotropic media. Myocardial tissue has an anisotropy resulting from the preferred orientation of the long axis of the myocytes. This results in a ratio of conduction speeds in the axial (aligned with the long axis of the myocytes) and transverse direction of approximately 3:1 in ventricular tissue [69, 72]. We will now derive the modifications to our constraints necessitated by this behavior.

The fact of preferred fiber orientation requires only a slight modification of the arguments made in section 2.3. There is still no *discrete* or discontinuous preferred propagation direction, i.e. there are no "sharp corners" in the system as there would be on a square lattice. Conceptually, this preferred direction represents a *stretching* of the original lattice, so that the transverse direction "appears" longer to a propagating wavefront. Mathematically, this a modification from circular to *elliptical* symmetry.

We start by generalizing (2.5) to two Cartesian coordinates, with the scaler diffusion constant, D, generalized to the diagonal diffusion tensor $\tilde{D}$:

$$\tilde{D} = \begin{pmatrix} D_x & 0 \\ 0 & D_y \end{pmatrix},$$

where x and y represent the fast and slow propagation directions, respectively. Equation (2.5) becomes:

$$\frac{\partial V}{\partial t} = D_x \frac{\partial^2 V}{\partial x^2} + D_y \frac{\partial^2 V}{\partial y^2} - \frac{f(V)}{\tau_V}. \tag{2.44}$$

The components of the diffusion tensor are coefficients of the second-order terms. For plane waves, there is always a characteristic dimensionless speed, $\hat{c}_0$ that is related to the components of the speed by:

$$c_0^x = \hat{c}_0 \sqrt{\frac{D_x}{\tau_V}} \qquad c_0^y = \hat{c}_0 \sqrt{\frac{D_y}{\tau_V}} \tag{2.45}$$

so the ratio of speeds will be related to the ratio of the components of the diffusion tensor:

$$\frac{c_0^y}{c_0^x} = \sqrt{\frac{D_y}{D_x}} \equiv \lambda. \tag{2.46}$$

We now introduce a "stretched" y-coordinate, $\tilde{y}$:

$$\begin{aligned} \tilde{y} &= y\sqrt{\frac{D_x}{D_y}} \\ \frac{\partial}{\partial y} &= \sqrt{\frac{D_x}{D_y}}\frac{\partial}{\partial \tilde{y}} = \lambda \frac{\partial}{\partial \tilde{y}}, \end{aligned} \tag{2.47}$$

and rewrite (2.44) in the new coordinates:

$$\frac{\partial V}{\partial t} = D_x\left(\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial \tilde{y}^2}\right) - \frac{f(V)}{\tau_V}, \tag{2.48}$$

which recovers the circular symmetry in the rescaled coordinate system.

To determine neighborhood size as in section 2.5, we again examine the speed curvature relationship. The curvature for a circle is always defined as $\kappa \equiv -1/r$, where r is the radius of the circle. For an ellipse we can only define a curvature for the situation where propagation of the wavefront is parallel to an axis of the ellipse. We represent the $\tilde{y}$-component of the wavefront as a function of x, $\tilde{y} = g(x)$, and the second partial derivative (curvature) becomes an ordinary derivative in $\tilde{y}$:

$$\tilde{\kappa} = \frac{d^2}{d\tilde{y}^2} = \frac{d^2 g}{d\tilde{y}^2},$$

which implies, in the unstretched system:

$$\frac{d^2}{dy^2} = \frac{1}{\lambda^2}\frac{d^2}{dy^2} = \frac{1}{\lambda^2}\kappa, \tag{2.49}$$

so we can map from the stretched (circularly symmetric) to the unstretched systems to get the curvature in the stretched direction:

$$\tilde{\kappa} = \frac{D_y}{D_x}\kappa, \tag{2.50}$$

where $\tilde{\kappa}$ is the same as in the isotropic case. Since in the stretched system we have $c^x = c_0^x + D_y\tilde{\kappa}$. We can substitute (2.50) for $\tilde{\kappa}$ in each direction, and therefore generalize Constraint 2 as follows:

Constraint 3 *The axes describing the elliptical neighborhood of an element must be chosen such that small curvature waves propagate with speeds:*

$$c^x = c_0^x + D_y\kappa \tag{2.51}$$

$$c^y = c_0^y + D_x\kappa. \tag{2.52}$$

We therefore choose the semimajor axis, $R_>$ from (2.43) and choose the semiminor axis as $R_< = R_>/\lambda$, where $\lambda \equiv c_>/c_<$.

It is interesting to note that the first-order speed-curvature relationship for the x-direction depends on the y-component of the diffusion tensor. This is because the first-order correction to speed is based on diffusive flux *transverse* to the direction of wave propagation. For a plane wave, all current travels forward because any transverse current would break the symmetry of the wave (left-going current from one part of the wave would be canceled by right-going current from a neighboring part). For a curved wave, however, current diffuses both forward and transversely. The transverse diffusive flux, which is therefore not available for forward propagation, will be proportional to the *transverse* component of the diffusion tensor.

2.7 Modeling Tissue Inactivation

The recovery process described in section 2.1.2 for solitary waves is a simplification of myocardial behavior. While it provides information about the *shape* of the excitation wave, it fails to account for the fact that the shape is not the only determinant of its propagation properties. This is because of the process of *inactivation*, in which

the ability of the excitation wavefront to source current decreases much more rapidly than given by $g(V, u)$ in (2.17). This phenomenon has a strong effect on the behavior of high-curvature waves.

Inactivation is incorporated into the model as a first-order effect on wave propagation. Ion-channel simulations allow the inactivation of Na^+ channels to be "turned off" and therefore the effect inactivation to be isolated. Using these techniques, it can be determined that inactivation is responsible for a decrease in plane wave speed of approximately 10% [25]. This requirement provides a constraint on source power:

Constraint 4 *The excitation current sourced by an excitation wavefront must decrease with time to give a 10% decrease in propagation speed relative to the trigger wave case.*

This constraint is implemented in the model by implementing a wavefront source current that decreases linearly with time:

$$I(t) = I_0(1 - \gamma t) \qquad \text{for } t < 1/\gamma. \tag{2.53}$$

The calculation of γ is as follows. First, the trigger wave threshold is calculated by the method by the method of section 2.4, for a default trigger-wave speed, c_0, and a corresponding solitary wave speed, $0.9c_0$. The threshold for the solitary wave will be higher than for the trigger wave, and is taken to be the threshold area for the element. The ratio of the two areas is then taken to be average power delivered by the excitation wavefront during the first time step.

The average of (2.53) during the first time step, Δt is $I(\Delta t/2)$. Let P be the ratio of the solitary wave threshold area to the trigger wave threshold area at a given plane trigger wave speed, c_0:

$$P = \frac{A_s(0.9c_0)}{A_t(c_0)}. \tag{2.54}$$

We then have:

$$1 - \frac{\gamma}{2} = P \qquad \gamma = 2(1 - P). \tag{2.55}$$

To get a good first-order representation of inactivation, we therefore also require $\Delta t \lesssim 1/(2\gamma)$.

2.8 Conclusion

The mathematical constraints presented in the chapter are necessary but not sufficient for modeling the behavior of an excitable medium. At the most, they give relationships between the parameters used in the state transition rules and empirically observed properties of the medium such as plane wave propagation speed, diffusion constant, and inactivation rate. For myocardial tissue in particular, there is also much more information that is needed, such as the detailed behavior of action potential duration, the propagation speed anisotropy, and the recovery of excitability after the action potential. These properties depend on the timing of excitation in ways that are beyond the scope of this chapter and are best modeled empirically.

Indeed, one of the strengths of the finite-state cellular automata approach used here is the ability to provide mathematic rigor as described in this chapter to ensure faithful reproduction of physical law, while still allowing the incorporation of complex empirical relationships. The exact model used in this study is a synthesis of the mathematical constraints described here and empirical observations, as described in chapter 3.

Chapter 3

Model

In sections 2.2 and 1.4.3, I described the general characteristics and some specific examples of models implemented as finite-state cellular automata. In this chapter I will describe in detail the model on which this study is based.

The heart is a macroscopic body. The paths traversed by excitation wavefronts are macroscopic phenomena. Yet the lesson of chapter 2, and of previous finite-element simulations that have given artifactual results [33,40], is that macroscopic phenomena will not be faithfully reproduced if microscopic physical interactions are distorted. This does not mean that it is necessary to reproduce *all* microscopic phenomena, but it does require attention to microscopic detail to ensure that critical phenomena are represented. In section 3.1, I will start with a description of the microscopic behavior of myocardial tissue. That will give enough material so that, in section 3.2, I will describe the construction of the computer model used for this study.

Given appropriate microscopic tissue behavior, sustained arrhythmia depends strongly on the geometry in which the aberrant excitation wavefronts propagate and interact. This interaction depends not only on the size of the macroscopic substrate but also on the geometry and topology seen by the excitation wavefronts. In section 3.3, I will discuss the macroscopic considerations involved in constructing a

realistic model of a ventricle.

Finally, one of the goals of this study is to evaluate pacing. The exact details of the interaction of electric fields with myocardial tissue will not be crucial to the questions considered, but it will be necessary to develop useful abstractions about the external programmed stimulation of myocardial tissue. In section 3.4, I will describe the model used to allow pacing of the substrate and discuss the level of information that the pacing model provides.

3.1 Additional Tissue Properties

Myocardial tissue is an excitable medium, and the general properties of excitable media are discussed in sections 2.1 and 2.2.1. These properties are only a subset of the important properties of myocardial tissue. All excitable media has a recovery period, but in myocardial tissue, as opposed for example to nerve tissue, the recovery (or refractory) period is very long compared to other time scales in the system. The length of the refractory period has a significant impact the ability of the tissue to support arrhythmia. The duration of the refractory period depends on the excitation history of the myocardial tissue. Higher stimulation frequencies are associated with shorter refractory periods. This phenomenon is called *restitution*, and is fundamental to the understanding of wave phenomena in myocardial tissue.

In section 2.4, we saw that wave propagation in excitable media could be characterized by a plane wave speed, c_0. In myocardial tissue, this speed is variable and depends on the stimulation frequency. Specifically, tissue can be excited while still in the relative refractory phase of the action potential, but the plane wave speed at which the excitation propagates will be lower than if the tissue were allowed to recover completely. The relation between the speed, c, and the local interbeat interval, T, is often referred to as the "dispersive property" of the tissue. This choice of language

is due to the resemblance between $c_0(t)$ and the dispersion relation between speed c and wave period, T, in linear wave theory. Again, understanding (and modeling) of this property is essential to faithful reproduction of arrhythmia.

Restitution and dispersion also vary spatially in the heart. The refractory time (action potential duration) will vary depending on external conditions, especially ischemia [34]. As a result, the duration of the action potential may be pathologically increased or decreased in distinct regions of the heart or in the heart as a whole. This condition (dispersion of refractoriness) has long been suggested as a crucial predisposing factor to arrhythmia [7, 37, 45, 51, 59, 62].

In this chapter, I will first present a detailed description of the restitution and dispersion properties of myocardial tissue. These properties, along with the fundamental properties of excitable media described in chapter 2, will form the basis of the modeled myocardial behavior. This will allow the complete specification of the rules for for the finite-state cellular automata model of myocardial tissue used for this study.

3.1.1 Restitution

Restitution decreases the duration of the action potential at higher excitation rates. On the electrocardiogram, action potential duration is associated with the interval between the 'Q-wave,' which results from first depolarization of the ventricle, and the 'T-wave,' which results from repolarization: the 'Q-T interval.' Restitution therefore accounts for the clinically-observed fact the Q-T interval decreases with heart rate (see, for example [53]). A commonly used clinical relation (the "Bazzett formula") between Q-T interval and heart rate give the expected "corrected" Q-T interval, Q-T_c as,

$$\mathrm{QT}_c = \frac{\mathrm{QT}_0}{\sqrt{\mathrm{R\text{-}R}}}, \tag{3.1}$$

where R-R is the time between successive depolarizations (in seconds),[1] and Q-T_0 is taken to be 0.42 sec in a normal patient.

Restitution is beneficial hemodynamically because the passive filling of the ventricle takes place while the ventricle relaxes during diastole, which starts near the end of the action potential. If action potential duration remained constant, a decrease in the time between successive heartbeats would result in the same decrease in diastolic filling time, and therefore filling would effectively vanish in a normal person for periods below 300 msec (a heart rate above 200 bpm)[2]. Restitution causes some of the decrease in time between heartbeats to be at the expense of *systole* (during the action potential), which is appropriate since a shortened filling time allows a shortened ejection time.

Restitution also provides some protection electrophysiologically. As we will show in section 4.5, there are mechanisms which result in the decrease of myocardial excitability near the end of the action potential. An excitation in low-excitability myocardial tissue is associated with higher probability of arrhythmia from several mechanisms (see, for example, Sec. 4.6), and accounts for the clinically-observed fact that tachyarrhythmia is often associated with an "R-on-T" in the electrocardiogram—a ventricular depolarization during the period of ventricular repolarization. Restitution, by decreasing the duration of the action potential, increases the time between the end of the previous action potential and the beginning of the next, thus decreasing the probably that a rapid beat will result in "R-on-T"-generated arrhythmia.

Once tachyarrhythmia has developed, however, restitution may be disadvantageous. One of the mechanisms that protects the heart against *reentrant tachycardia* is the refractory period, which prevents rapid reexcitation by a reentrant path. The shortening of action potential duration, and therefore the refractory period, de-

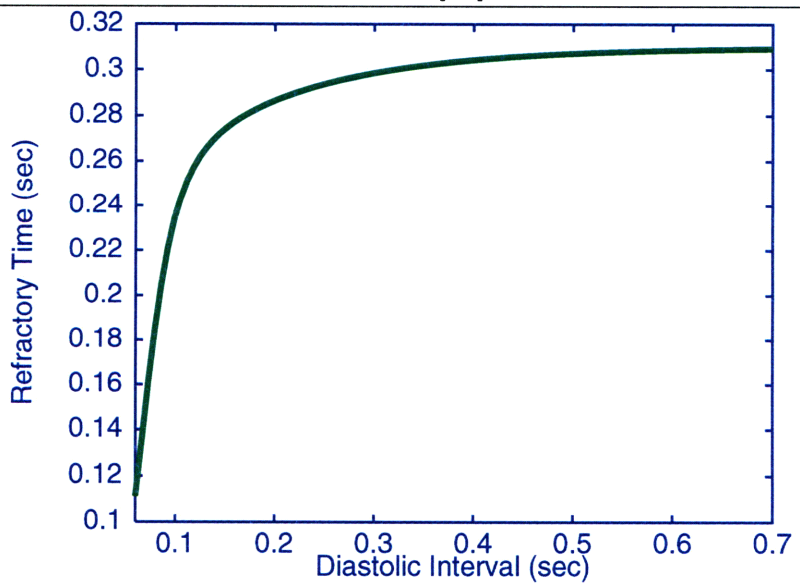
[1] On the electrocardiogram, the 'R' wave is usually the main deflection resulting from ventricular depolarization, and the distance between R-waves therefore represents the best measure of the excitation period.

[2] This number is based on the "standard" action potential duration of 300 msec, given by [43]

creases the protection provided by this mechanism. At high stimulation frequencies, the refractory period becomes very short, effectively perpetuating the problem [6,68]. Restitution may also be responsible for alternation of the action potential duration, a condition called *action potential alternans*, which may be a precursor to cardiac sudden death [18,42].

Equation (3.1) is clinically useful for the "normal" range of heart rates, but is inaccurate for the extreme stimulation frequencies seen in reentrant tachyarrhythmias. There have been many studies of the details of the restitution relationship in humans and animals [6,29,31,36,68]. The data from these studies generally show a shortening of action potential duration by approximately 40–60%, asymptotically approaching a "nominal" value for cycle lengths on the order of 1 second. Restitution also depends to some extent on the "steady-state" rate of excitation, but there is little comprehensive quantitative work on this. There have been some studies suggesting a further role, in some cases, of bradycardia in prolonging action potential duration, though this may be associated with other changes in the action potential as well [70].

Figure 3-1 Resititution Relationship from [20]



For this study, it was necessary to have a quantitative relationship with a simple

protocol (i.e. independent of the details of pacing electrodes and AV/His propagation parameters). Such a study was performed by Courtemanche, *et.al.* [20] on a one-dimension Beeler-Reuter [11] simulation and provided the following restitution relationship:

$$\mathrm{APD}(t_r) = 0.02 + \left[0.25 - 0.09 \exp\left(\frac{-t_r}{0.145}\right)\right] \frac{t_r^{5.5}}{0.072^{5.5} + t_r^{5.5}}, \tag{3.2}$$

where APD is the action potential duration in seconds and t_r is the previous diastolic interval (recovery time) in seconds (Figure 3-1). While this equation provides little insight into the mechanisms governing restitution, it has the advantage of being a continuous function which is based on ion-channel studies, and it gives results similar to empirical studies.

3.1.2 Dispersion

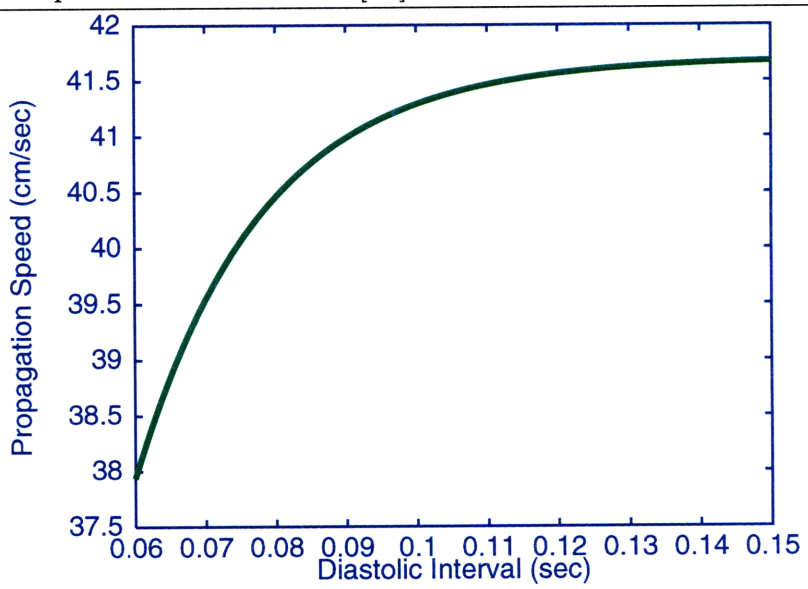
Near the end of the action potential, tissue enters a period of "partial excitability" or "relative refractoriness." As the terms imply, tissue in this state may be stimulated, but it requires a stronger stimulation to excite (it is more refractory)[3]. Like restitution, this effect has significant implications for propagation of excitation wavefronts. Since tissue which is less excitable has a lower plane wave propagation speed, an excitation wavefront propagating in partially-excitable tissue will propagate more slowly. The resulting relationship between excitation period and propagation speed is called *dispersion* (not to be confused with "dispersion of refractoriness").

Dispersion can play a significant role in creating pathways for reentrant arrhythmia, since the "effective length" of a pathway is longer for slower propagation (see also section 4.1 for a discussion of *action potential wavelength*). At least as important as propagation speed is the fact that excitation wavefronts in lower-excitability tissue

[3]Certain specialized myocytes go through a "Supernormal" period immediately after the end of the action potential in which they are *more* excitable, but this does not appear to be an important effect in ventricular tissue [43]

interact very differently with obstacles. Experimental and theoretical studies have shown arrhythmogenesis even in healthy tissue when it is stimulated during a period of decreased excitability [32, 66], and I consider this effect in detail in section 4.4.

Figure 3-2 Dispersion Relation from [20]



The change in excitability from dispersion is most conveniently quantified as a decrease in propagation speed. The same numerical study that provided the restitution relationship (Sec. 3.1.1) gives the following dispersion relation [20]:

$$c_0(t_r) = 41.7 - 13.5 \exp\left(-\frac{t_r - 0.037}{0.018}\right), \tag{3.3}$$

where c_0 is in cm/sec and t_r is the previous diastolic interval (recovery time) (Figure 3-2). The results from this study were qualitatively similar to experimental studies of canine tricuspid rings performed by Frame [29], though the nominal propagation speed of 42 cm/sec is below the generally accepted value for human ventricular myocardium.

Again, since the shape of the curve is close to experimental studies, and since it provided a continuous relation based on theoretical studies, I based the dispersion relation in the model on this function, after renormalizing the default speed to 50 cm/sec, which is more commonly taken to be the plane wave speed in ventricular myocardium [72]. Since (3.3) and (3.2) are products of the same study, and

because *dispersion* and *restitution* are strongly coupled phenomena, the use of either relationship is a good justification for using the other.

3.2 The Finite-State Cellular Automata Model

The specific tissue properties described in section 3.1 and the general properties of excitable media described in chapter 2, provide the basis of our model of myocardial tissue. In this section I will give the rules which completely determine the behavior of the finite-state cellular automata model used subsequently in this study.

3.2.1 Discretization of State Space

In section 2.2 I presented the minimal set of states required to faithfully model excitable media: RESTING, in which an element may be excited by its neighbors, EXCITING, in which an element may cause excitation of its neighbors and REFRACTORY in which an element may neither excite nor be excited by its neighbors. To provide a more complete model of myocardial tissue, it is necessary to add one more state to this set: RELATIVE REFRACTORY, which is characterized by *partial* excitability, i.e. excitement is possible but requires a higher level of stimulation that for resting tissue. This state corresponds to the *relative refractory* period described in section 1.1.1. The reduced excitability in the RELATIVE REFRACTORY state is associated with a decreased wave propagation speed following excitation.

In fact, the term *state*, as used here, is inaccurate. The state of an element is the total of all internal information which controls the evolution of that element. Therefore, if the REFRACTORY state has a specific time duration, the "state" of the element also includes the value of the timer that will determine the expiration of the refractory period. In this model, each of the four "states" described has a finite duration and transition is therefore controlled in part by a timer. For simplicity,

however, I will adhere to the common and more intuitive usage of the word "state" to mean a broad characterization of the types of transitions possible for an element, and deal with the timers separately. By this usage, "states" should not be considered *points* in state space but more approximately *regions* in state space.

Figure 3-3 shows a schematic of the states used in the simulator and the conditions under which transitions between the states occur. Each state is described in detail below.

The RESTING State

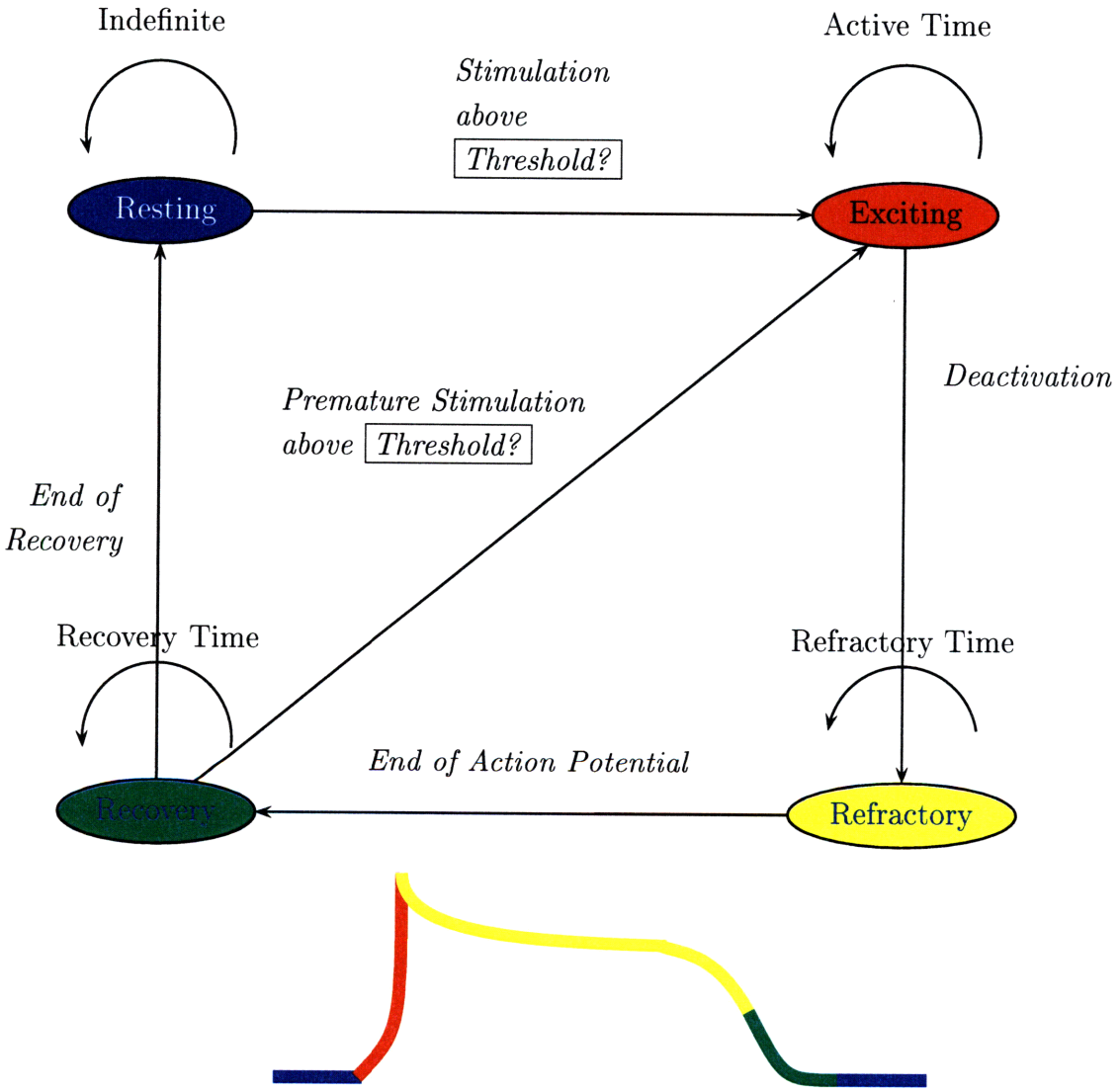
Tissue at rest tends to remain at rest—RESTING is the only stable state. The RESTING state corresponds to tissue that has been repolarized for a sufficiently long time to have recovered full excitability (Sec. 3.1.2). RESTING tissue remains in that state indefinitely until it is excited.

As described in Chapter 2, each element has a threshold and a neighborhood. The threshold of a resting element is calculated from (2.4), based on Constraint 1, to give a plane wave speed of c_0, which is normally taken to be 50 cm/sec in the "fast" propagation direction (Sec. 2.6) for normal ventricular tissue. The neighborhood of an element is an ellipse calculated from (2.43) and Constraints 2 and 3.

The EXCITING State

Only tissue in the EXCITING state is capable of effecting the state of its neighbors. When an element is excited, either from RESTING or from RELATIVE REFRACTORY (below), it enters the EXCITING state. For each time step in which the element remains in this state, the element "delivers" source current to each element in its neighborhood. The amount of source current delivered declines linearly with time according to (2.55) and Constraint 4. The total source current delivered to an element

Figure 3-3 States and state transition rules for the CA simulator. The four principle states are shown, with long arrows showing the possible transitions and the rules for those transitions. Arrows connecting a state with itself represent a timer which controls an automatic transition to the "next" state, except for the resting state which has an indefinite lifetime. For comparison, an illustration of the action potential is given below. The colors of the states refer to the colored regions of the action potential



is then compared to the excitation threshold for that element. If it exceeds the excitation threshold, that element will also enter the EXCITING state. When the source power that an EXCITING element would deliver during the next time step declines to zero, the element enters the REFRACTORY state.

The REFRACTORY State

Elements in the REFRACTORY state are completely autonomous: they can neither affect nor be effected by any of their neighbors. The time an element spends in the REFRACTORY state is given by the current action potential duration for that element. In general, this will vary with both the location of the element (dispersion of refractoriness) and its excitation history (restitution). When the element enters the REFRACTORY state, the duration of that state is determined based on its previous diastolic interval (Equation 3.2) and its nominal refractory period. When the timer associated with this state has expired, the element enters the RELATIVE REFRACTORY state.

The RELATIVE REFRACTORY State

RELATIVE REFRACTORY is the intermediate state between REFRACTORY and RESTING. Elements in this state can be excited, but their excitation requires more source current that for RESTING elements. The source current required (the local *excitation threshold*) decreases to the nominal (RESTING) value according to the dispersion relation (3.3): at each time step, the speed at which an excitation wave *would* propagate for that element is calculated from (3.3), and the excitation threshold for this speed is then calculated from (2.4) as for RESTING elements. If a RELATIVE REFRACTORY element is not excited, it will enter the RESTING state when the timer associated with the RELATIVE REFRACTORY state has expired. Unlike the REFRAC-

Table 3.1 States defining the finite-state cellular automata

State	Characteristic	Transition
RESTING	Element is fully excitable at nominal propagation speed	External excitation: $\longrightarrow$ EXCITING
EXCITING	Element has depolarized and is capable of depolarizing neighboring elements	Inactivation: $\longrightarrow$ REFRACTORY
ABSOLUTE REFRACTORY	Element cannot be excited	End of Action Potential: $\longrightarrow$ RELATIVE REFRACTORY
RELATIVE REFRACTORY	Element can be excited but only at reduced propagation speed	recovery $\longrightarrow$ RESTING **OR** External excitation $\longrightarrow$ EXCITING

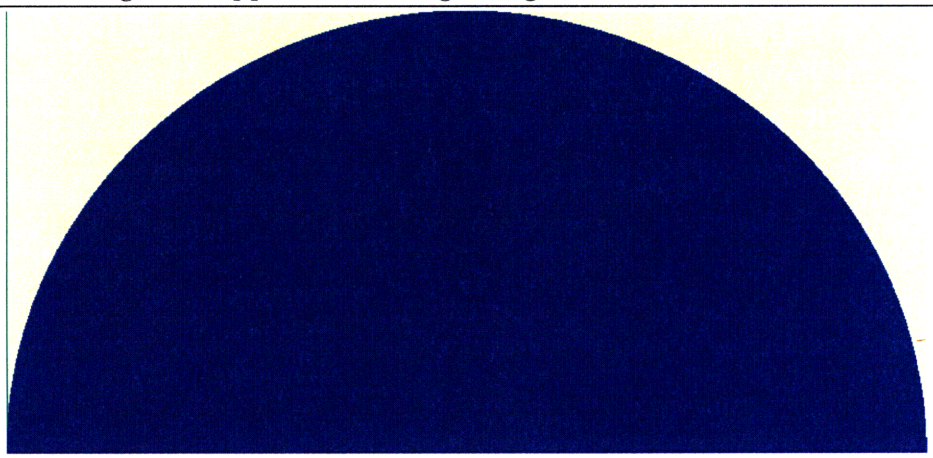
TORY state, the RELATIVE REFRACTORY state has a constant duration, independent of location or history.

These states and state transition rules are summarized in Table 3.1. The power of finite-state cellular automata is that these relatively simple and intuitive rules *completely* define the microscopic system and therefore represent our microscopic understanding of the system. All the complex behavior that will be seen later in this study are consequences of these rules.

3.3 The Macroscopic Substrate

Appropriate microscopic interaction provides the character of excitation wavefronts, but the substrate geometry determines their interaction. The two properties of the macroscopic myocardial system which must be modeled are the geometry of the system: the size and shape that determine the paths available to the wavefronts, and tissue anisotropy, which determine the speeds at which the wavefronts propagate on these paths. Section 3.3.1 discusses the considerations of size and shape, and sec-

Figure 3-4 The projection scheme for the simulated heart. The entire heart is in the RESTING state (blue). This projection shows the apex on top and the atrio-ventricular annulus at the bottom. The right and left edges are connected so a wave running "off" the left edge will appear on the right edge.



tion 3.3.2 discusses the importance and modeling of propagation speed anisotropy.

3.3.1 Geometry

Many of the simulations done in this study were "whole-ventricle" simulations designed to simulate ventricular arrhythmias. The model geometry for these simulations was chosen to approximate actual ventricular geometry as closely as possible within model limitations. The lattice was a hemi-spheroid with a circumference of 21 cm and a base-to-apex length (arc length) of 10 cm. These numbers were chosen to give a reasonable compromise between a juvenile swine heart (a commonly-used experimental model) and a "normal" adult human epicardium. Since there is a wide variation in the sizes of hearts which are susceptible to arrhythmia, we wouldn't expect the model or the results to be critically sensitive to the choice of size. We require only that the dimensions be large enough to support multiple reentrant paths.

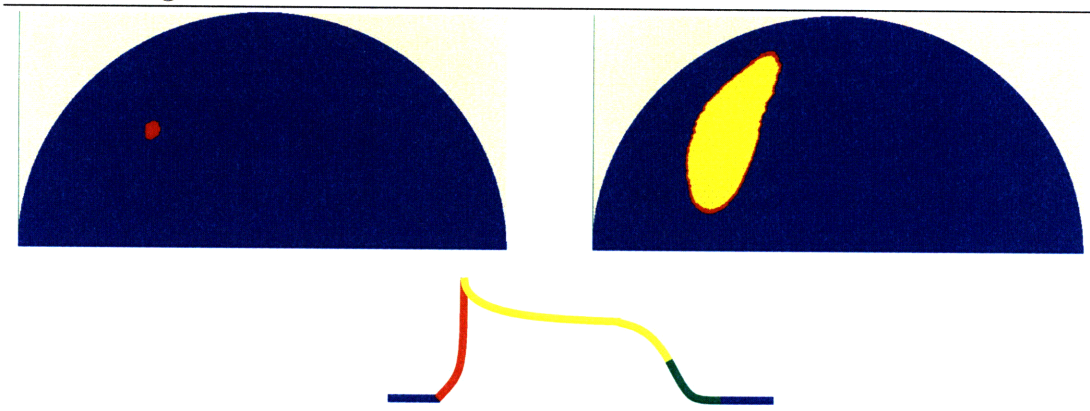
The hemispheroidal geometry was used to give a good approximation to the topology of the ventricles. Many two-dimensional models have used the simpler cylindrical geometry [18, 52, 60, 63]. In this study we chose hemispheroidal geometry because

it accurately represents the boundary at the AV annulus (the base) and the closed surface at the apex, allowing apical waves to "wrap over" the apex, but forcing basal waves to move circumferentially, as would happen in the heart. The lack of a ventricular septum is probably less important. Unlike the apex, the septum only provides a "short-cut" between the walls of the ventricle, decreasing the path length by perhaps 30%. The apex, in contrast, provides motion in an completely independent direction, i.e. a wave near the apex can quickly "get to the other side" by going over the apex, a path that isn't available on a cylinder. This changes the interaction of waves near the apex and also allows stable waves at any angle to the axis of the heart.

The lattice was rectangular, so implementation of the overall hemispheroidal geometry required connectivity that was locally not one-to-one. It was implemented as a set of horizontal circles (stripes), whose length (circumference) was a function of their y-position. Adjacent stripes generally had different numbers of elements. The deficit was reconciled by randomly doubling up-down links from the shorter to the longer circle, meaning that some adjacent elements in the longer stripe would each point to the *same* element in the shorter stripe, though the element in the shorter circles would obviously point back to only one of the elements in the longer circle. This was done to ensure smooth and symmetric propagation of waves in all directions on the lattice. For a complete description of this process see the comments and code in A.7, near line 350.

The lattice was projected on the screen with each loop "flat" and the loops stacked horizontally (Figure 3-4). Opposite sides at the same vertical level are continuous (periodic boundary conditions in x). This was the simplest projection scheme and did not distort the shapes of the waves.

Figure 3-5 The effects of anisotropy of propagation speed on a circular pulse. The frame on the left is a small circular region of excitation. The frame on the right is the resulting excitation wave 60 msec later.



3.3.2 Tissue Anisotropy

Ventricular anisotropy results from the fact that cardiac myocytes are approximately cylindrical, with an axis of approximately 100 μm and a diameter of only approximately 10 μm. Due to the spatial distribution of gap junction connections and the geometry of the cells, the bulk conductivities and thus the diffusion constants differ by a factor of 10 for axial and transverse propagation. As explained in Sec. 2.6, the results in a propagation speed which approximately 3 times greater in the axial dimension (Figure 3-5).

Tissue anisotropy is important for several reasons in understanding the arrhythmogenesis. First, the existence of a slow propagation direction changes the "effective" size of the chamber in that direction. A wavefront traveling in the direction of slow propagation will "see" a longer distance to travel, since it will take longer. This is important because it has the effect of increasing the length of portions of a potentially-reentrant pathway, making the pathway long enough to support sustained reentry.

Second, wavefront-obstacle interactions often lead to *separation*, in which the wavefront detaches from the obstacle and pursues an independent course. Such a wavefront can be arrhythmogenic, but *only* if it remains near the obstacle long enough

for surrounding tissue to recover (section 4.6). In practice, this means that only separation of wavefronts moving in the fast direction is commonly arrhythmogenic, because this allows the transverse wavelet to travel in the slow direction.

Finally, there are studies suggesting that part of the mechanism for arrhythmogenesis involves changes in the anisotropy of the medium. It is therefore useful to be able to model anisotropy, though this effect may be limited to three-dimensional systems. [65]

The difficulty in modeling anisotropy in a two-dimensional system is that the fiber orientation is known to rotate continuously from epicardium to endocardium [43, 69, 72]. This clearly cannot be modeled in a two-dimensional system where there is no component transverse to the chamber wall, so it was necessary to choose an orientation direction. For this study, I chose the axial direction, rather than the circumferential direction, which is the approximate orientation on the surface of the endocardium [72].

The model should not be very sensitive to the choice of the fast propagation direction, since it will still satisfy the first two criteria discussed above necessitating an anisotropic model. Orientation probably is significant for accurately modeling infarcted regions close to the AV annulus, however, since an axial orientation (as chosen for this study) will slow the direction through the "one-dimensional" pathway between the infarction and the annulus. The chose of an endocardial orientation will therefore consistently model endocardial excitations (as for normal HIS mediated beats) or endocardial PVCs in the context of transmural infarctions.

3.4 Pacing

Pacing myocardial tissue involves the interaction of an external electric field with the mycardial membranes. The field can be applied with small electrodes attached to the

heart (usually for pacemakers), specialized catheters (for electrophysiologic testing), large patches or coils (usually for ICDs), or even external electrodes (for external cardioverters and transcutaneous pacemakers). The efficacy of any of these systems in exciting a cell depends on the excitability of the cell and the voltage drop developed across the cell membrane. This depends in turn on the total voltage applied across the electrodes, the distance separating them, and the resistance of the tissue between the electrodes and the myocytes. Tissue far from the electrode is not usually excited by low-voltage pulses. Therefore larger electrodes will often excite more tissue than small electrodes, but increasing the electrode voltage or the width of the stimulation pulse will also generally increase the amount of tissue excited.

In this model, we will not be concerned with exact details of electric field. No attempt is made to model the interface of the electrode to the tissue, nor the geometry of field. There are several reasons for this. First, the issues of exact field shape for a particular electrode configuration, given the details of intervening tissue, are still an area of extensive research and beyond the scope of this study. Second, the considerations that lead to the choice of a two-dimensional model preclude the modeling of exact field geometry. The effect of the field changes at different depths in the ventricular wall. In a two-dimensional model, there is no transmural component, so an attempt to create a realistic model of field inhomogeneity would quite probably be an over interpretation of the model.

Most importantly, however, is the consideration of simplicity described in the introduction to chapter 2 regarding modeling in general. Our goal here is to develop a clean and intuitive model of pacing and arrhythmia so the results can be interpreted. Depending on the results of these studies of pacing it may be advantageous to implement a more sophisticated electric field model in the simulations.

Pacing a cellular-automata model is equivalent to the selection of particular elements for excitation. That selection will be based on two criteria:

- The location of the element. In general a pacing impulse is applied to a specific region of the heart. The corresponds to an electrode of finite size and a relatively low-voltage pacing pulse (a few times *diastolic threshold*—the voltage at which a fully-recovered region of tissue will be excited)[4]. Elements that are not within the stimulated region will not be excited.

- The current excitation state of the element. A RESTING element can always be stimulated by a pacing pulse. As an element recovers during the RELATIVE REFRACTORY state, its excitation threshold decreases. In this model the "pacing strength" will always be specified in terms of that excitation threshold. Elements whose excitation threshold is less than the "strength" of the pacing pulse can also be stimulated.

These criteria allow the specification of each pacing stimulus as follows:

> A particular pacing stimulus will result in the excitation of all elements within a region specified for that stimulus that have recovered to at least the excitability specified for that stimulus.

This ability to specify the regions stimulated and the excitability of the tissue in which the resulting excitation waves will propagate provides a very general abstract model of the *results* of pacing, without the requirement of incorporating the *mechanisms* of pacing.

3.5 Conclusion

Much sophistication was involved in determining the constraints specified in chapter 2 which control a finite-state cellular automata model, and in determining the

[4]High-voltage pulses are used for cardioversion and are designed to create a voltage many times greater than the voltage at which excitable myocytes would be stimulated, and to create that voltage even for those myocytes which are most distant from the pacing electrode [39]. We will make no attempt to model pacing pulses of that strength.

additional tissue properties discussed in section 3.1. The resulting model, however, remains comparatively simple. This allows relatively intuitive analysis of the simulation results. Because of this, model will provide a powerful abstraction for the study of arrhythmia and pacing in the next chapter.

Chapter 4

Modeling of Arrhythmia

The model described in chapter 3 is designed to reproduce electrical phenomena at length scales varying from less than 1 mm to tens of centimeters. It can therefore provide meaningful results for a large range of electrophysiologic substrates. In this chapter I will use results of simulations on this model to discuss several mechanism for the generation of arrhythmias and demonstrate the importance of various types of electrical inhomogeneities ("obstacles"). These obstacles will have different effects on the propagation of excitation wavefronts depending on their length scales, but will all be important for understanding the mechanisms by which premature beats generate arrhythmia and by which those arrhythmias may be successfully, or unsuccessfully, terminated by paced beats.

I will show several examples of these electrical phenomena and describe their significance in the initiation, maintenance and termination of arrhythmia. In section 4.1, I will introduce the important length scales for wavefront obstacle interaction and demonstrate how local interactions with obstacles can lead to the formation of reentrant arrhythmia.

A reentrant arrhythmia is one in which an excitation wavefront traverses a path more than once, so that it "reenters" its own wake. Production of a reentrant arrhyth-

mia requires a disturbance in the propagation of the excitation wavefront, causing it to "turn around" and move counter to its initial direction ("retrograde"). In practice this always implies that there is a path that is preferentially traversed in one direction. One of the mechanisms frequently advanced to account for this behavior is *unidirectional block*, in which a section of tissue is unexcitable when a wavefront passes in one direction but has recovered excitability when it passes in the other direction [43]. Another mechanism that can account for reentry is *wavefront-obstacle separation*, where an excitation wavefront separates from an obstacle and curls into the excitable region left behind. Both these phenomena initially occur in small regions of tissue. If the arrhythmia becomes sustained, the resulting reentrant wavelets can then generate the dominant electrical behavior of the heart, effectively entraining all other activity. I begin, therefore with a detailed study of the local behavior that gives rise to arrhythmia and the length scales that control this behavior.

4.1 Length Scales in Myocardial Tissue

There are many length scales in myocardial tissue which are important to various phenomena. Characteristic sizes of ion channels, separations between ion channels and distances between cells are important for understanding the microscopic nature of trans-membrane ion transport and may be of interest, for example, to protein chemists involved in rational drug design. But even ion channel models operate at length scales larger than this so that these dimensions are effectively averaged out.

The smallest length scale directly modeled in cardiac electrophysiology is the size of the cardiac myocyte. This is because the action potential propagates somewhat discontinuously from cell to cell through specialized gap junctions called intercalated disks. A propagation model concerned with detailed membrane behavior therefore must operate at a length scale just below the myocyte size to allow distinguishing

between propagation along a single membrane and propagation through intercellular junctions.

For macroscopic electrophysiology—the propagation of excitation wavefronts in heart tissue—which is the realm of interest in this study, there are three important length scales. The most important length scale for analyzing the interaction of a wavefront with an obstacle is the width of the excitation wavefront or *transition region* between the RESTING and EXCITED states of the tissue (the *transition region wavelength*). This length is roughly equivalent to the spatial width of the rapid upstroke part of the action potential and is $\sim 1\,\text{mm}$. This is the "wavelength" that measures the size of an obstacle[1]: the effect of objects with size $\gg 1\,\text{mm}$ is very different from the effect of objects with size $\lesssim 1\,\text{mm}$ (see section 4.2, and figure 4-1).

The length scale most important for reentrant arrhythmia is the entire spatial extent of the action potential from excitation to recovery (the *action potential wavelength*). Any reentrant path in the heart must be at least this long because any region of tissue in the path must have sufficient time to recover before the next time that region is excited by the wavefront. This length will vary substantially with action potential duration and local propagation speed, both of which vary, in turn, with stimulation timing (Sec. 3.1.2 and Sec. 3.1.1), which is one of the reasons that the timing of a PVC is so important in arrhythmogenesis.

The final length scale of importance for our study is the size of the chamber. For reentrant arrhythmia, this must obviously be at last as large as the *action potential wavelength* to sustain the arrhythmia. But this length scale is also a main determining factor in whether a sustained reentrant arrhythmia becomes lethal. The significant difference between *ventricular tachycardia* and *ventricular fibrillation* is the presence of synchrony of contraction. In *ventricular tachycardia* there is sufficient synchrony

[1]The transition region wavelength has a close analogy to the wavelength of light: Light interacts very differently with objects less than it's wavelength (they are very hard to see even under a microscope) than with objects much greater than its wavelength

that the heart still contracts synchronously and there is some pumping action.

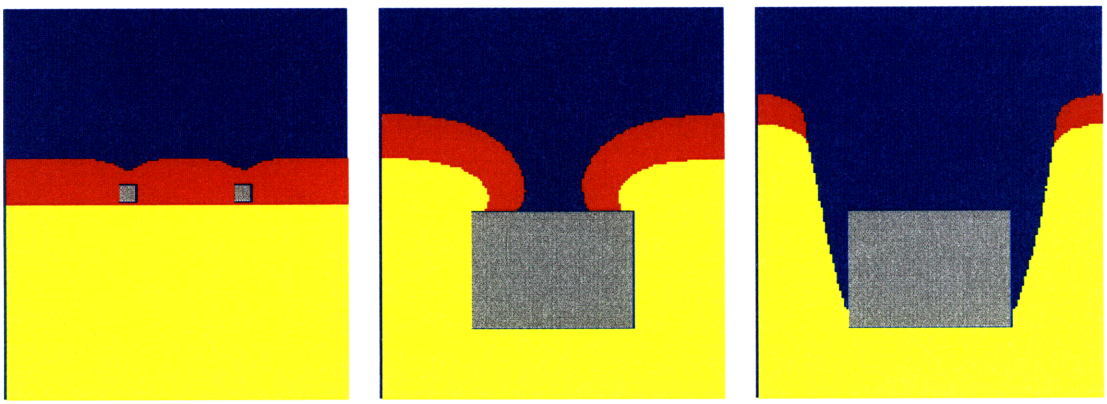
In *ventricular fibrillation*, synchrony is lost, resulting in catastrophic hemodynamic collapse. In the case of fibrillation, this is due to the presence of multiple reentrant paths in a single chamber which are not strongly correlated with one-another. Therefore, for a *ventricular tachycardia* to degenerate into *ventricular fibrillation* and thus sudden cardiac death, the tissue available for reentrant propagation should probably exceed several times the *action potential wavelength.*

There are two consequences of this constraint on the development of VF. First is the well-known fact that small animals do not appear to develop fibrillation since their hearts are too small to support multiple independent pathways. This also suggests that humans with very large ventricles secondary, for example, to dilated cardiomyopathy, would be expected to be more susceptible to the development of VF. Second, one can speculate that it is possible to *prevent* the degeneration of an organized reentrant arrhythmia such as VT into a disorganized and fatal arrhythmia such as VF by reducing the tissue *available* for conduction, thus reducing the effective chamber size below the threshold which can support VF. This suggests the possibility of preempting the transition from VT to VF by pacing, and is one of the protocols attempted below (Sec. 5.2.3).

4.2 Interaction Between Wavefronts and Obstacles

Figure 4-1 shows three examples of the interaction between an excitation wavefront and fixed obstacles, illustrating the importance of length scale and tissue excitability. In each case, the length scale is determined by the wavefront transition region. Here, the transition region has a spatial extent of order 2 mm, and is the "length-scale" that determines how a depolarizing wavefront will interact with an obstacle. Specifically, if the wavefront interacts with an obstacle smaller than the size of the *transition*

Figure 4-1 Important features of the depolarizing wavefront. In panel (a), the wavefront interacts with obstacles (gray) smaller than the width of the *transition region* (red). The obstacles are engulfed by the wavefront with no significant change in the shape of the wavefront. In panel (b), the obstacle is larger than the width of the *transition region*, and results in *fractionation* of the wavefront (the wavefront is broken). The two wavefront *fragments* pass on either side of the obstacle. Since the wavefront remains *attached* to the obstacle, the wavefront fragments must rejoin at the distal edge of the obstacle. In panel (c), the wavefront has been *fractionated* by a large obstacle in tissue of reduced excitability. In this case the *fragments* have not remained attached to the obstacle but rather *separated* from it and now move independently. These *fragments* will not, in general, rejoin as did the fragments shown in panel (b).



region, it will tend to "flow over" the obstacle without appreciably changing shape (Figure 4-1(a)).

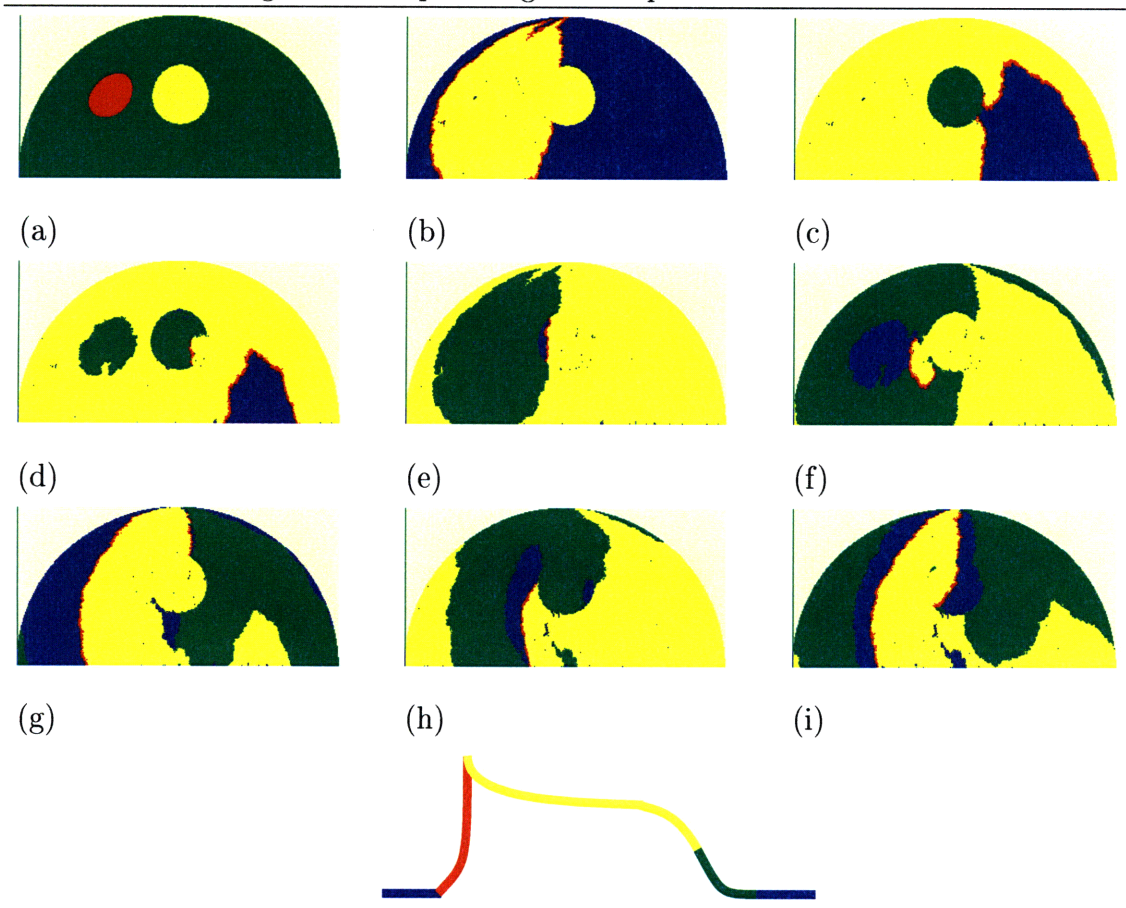
The situation is very different when a depolarization wavefront interacts with an obstacle larger than the size of the *transition region*. In this case, the oncoming wavefront is torn or *fractionated* (Figure 4-1(b)), meaning that the wavefront is no longer continuous, but has gaps which are greater than the width of the transition region. This distinction is important because *fractionation* is an essential precondition to reentrant arrhythmia.

Fractionation, though necessary, is not a sufficient precondition for the development of reentry via interaction with an obstacle. This is because wavefront fragments which remain attached to the obstacle will recombine into a continuous wavefront when the fragments have circumnavigated the obstacle (Figure 4-1(b)). The fractionation is therefore temporary and will not lead to reentry. Reentry requires the further condition of *separation*, where the wavefront *fractionates*, and the fragments then detach from the obstacle and move independently of each other. (Figure 4-1(c)). Since these *fragments* will not, in general, recombine, they have the potential to form reentrant circuits.

4.3 Unidirectional Block

Many models of reentry have proposed that *fractionation* is the result of the interaction of the depolarization wavefront with relatively large (larger than the width of the transition region) islands of recovering tissue, and that *separation* is the result of *unidirectional block* (Figure 4-2), in which the fractionating island regains excitability before the wavefront has passed it, and the wavefront *reenters* the newly recovered portion of the obstacle. The *fragments* of the wavefront are no longer attached to the object because the object (the refractory island) is disappearing. As the island

Figure 4-2 Arrhythmogenesis Caused by Unidirectional block. Panel (a) shows the starting conditions. most of the heart is partially recovered (green). There is a region in the center where the tissue has not recovered any excitability because of a longer than normal refractory period (yellow). A PVC has occurred close to this region (red). In panel (b) the PVC is propagating outward into fully-recovered tissue (blue). The region in the center is still refractory so the wavefront propagates around it. In panel (c), the region in the center has recovered partial excitability just as the wavefront passes the far side. The wavefront begin to penetrate backward into the central region. In panel (d), the wavefront propagates slowly across the partially-recovered region. In panel (e), the region on the left of the central region has recovered so the wavefront can propagate (retrograde) into it. In panel (f), the wavefront has broken out of the central region and is propagating outward into the heart. In panels (g) and (h), the (reentrant) wavefront is propagating around the central region as before. In panel (i) the now fully-reentrant wavefront has detached from the central region and is pursuing an independent course.



vanishes (after *separation*), the independent motion of the two wavefront *fragments* can, in a generic situation, lead to reentry. Thus the key elements for the development of reentry, even according to the conventional model, are *wavefront fractionation followed by wavefront-obstacle separation.*

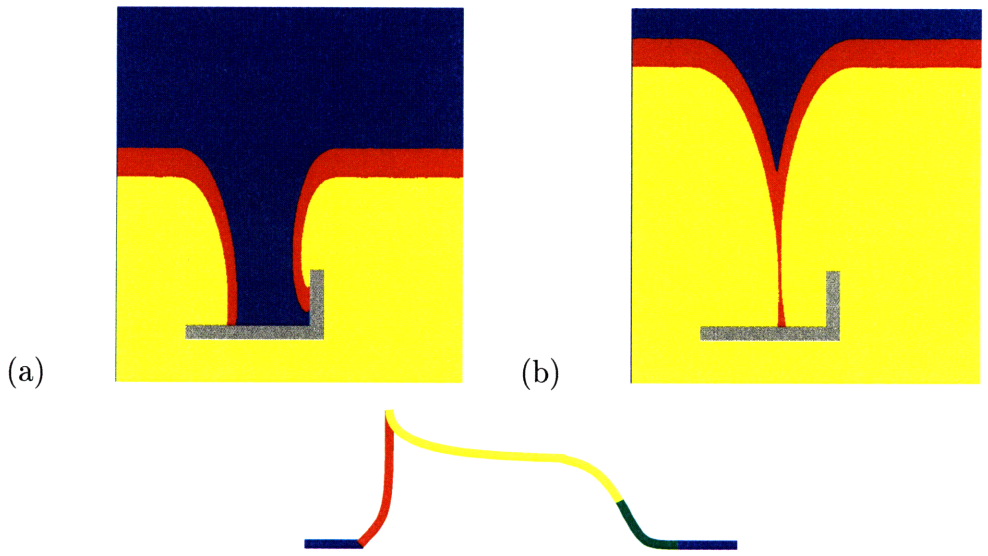
4.4 Wavefront-Obstacle Separation

In the unidirectional block scenario, separation occurs only because of the disappearance of the refractory island that originally caused the wavefront fractionation. As can be seen from figure 4-2, this will not lead to arrhythmia unless the size of the region having the extended refractory period is of the order of the *action potential wavelength.* If the region is too small, the wavefront will completely traverse the region before the outside section (which *was* excited by the normal wavefront) is able to recover (see figure 4-2, frames (d) and (e)). This requires not only a substantial amount of refractory tissue, but also that the refractory periods be closely correlated across a large region.

Arrhythmogenesis can occur through several other mechanisms that do not require such large spatial correlation lengths of tissue properties. The common step in all these mechanisms is *wavefront-obstacle separation*, where the excitation wavefront separates either from a fixed obstacle (such as a scar), or from a cluster of refractory tissue. Since separation does not therefore depend on the obstacle vanishing *exactly* as the wavefront passes it, there is a much larger range of objects which may be arrhythmogenic and a larger window of time during which arrhythmia may develop.

The basis of wavefront-obstacle separation is seen in figure 4-1 (c). The difference between panels (b) and (c) in this figure is the *excitability* of the tissue. In myocardial tissue of normal excitability (panel (b)) the ends of the wavefront fractions "stick" to obstacles. The consequences of this characteristic excitability can be seen in figure 4-

Figure 4-3 The interaction of a planar wavefront with a fixed, asymmetric obstacle in otherwise uniform, normally excitable myocardium. In panel (a), the wavefront is "attached," and closely following the distal contour of the obstacle. Panel (b) shows the later evolution of the wavefront. As a consequence of the attachment of the wavefront to the obstacle, the transversely-propagating portions of the two wavefront fragments collide and annihilate each other, leaving a single wavefront with a cusp that smoothes out as the wave propagates.



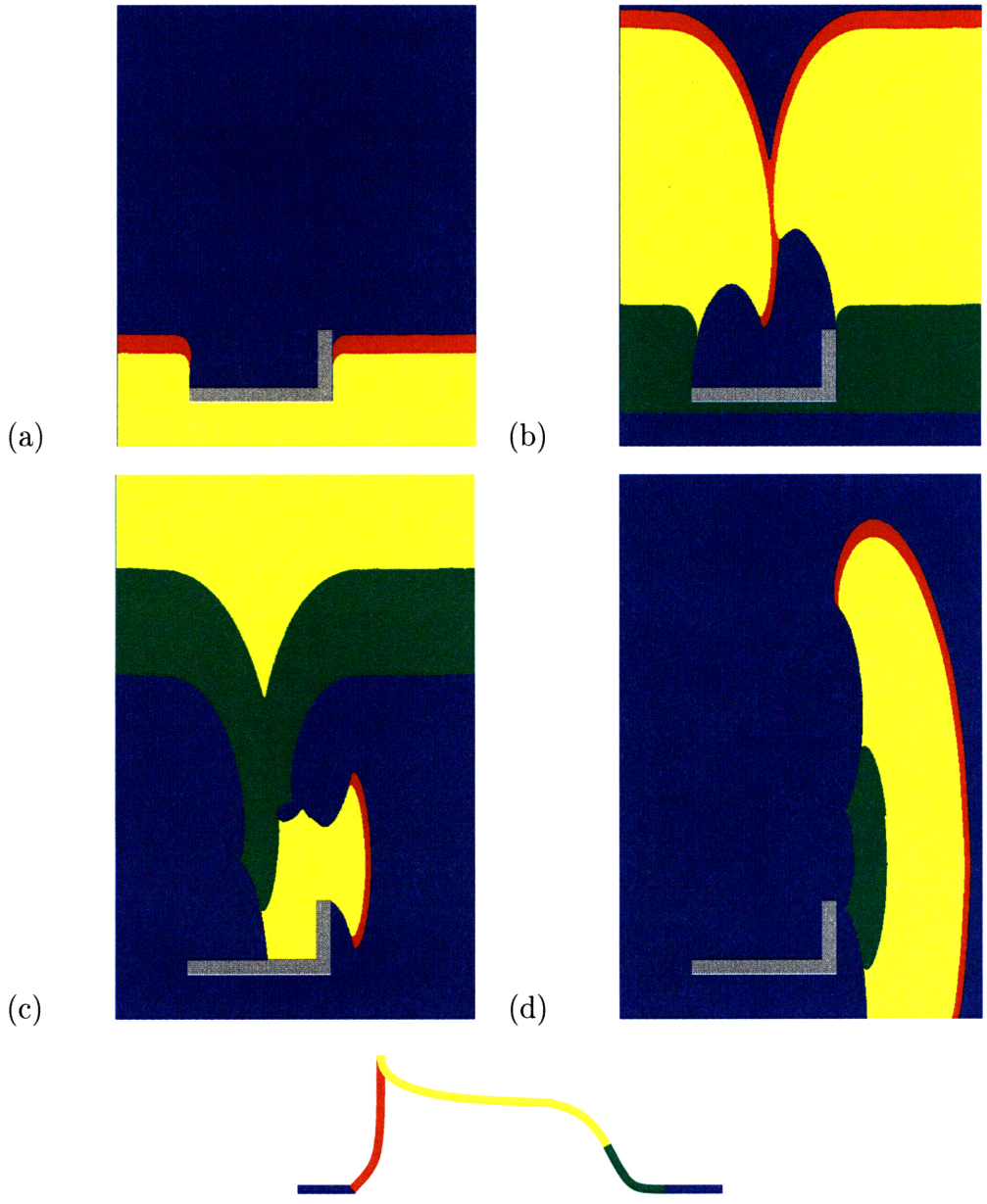
3. An excitation wavefront approaches an irregularly shaped fixed obstacle (scar) from below. The wavefront fractionates as it passes the obstacle, but because the tissue is normally excitable, the ends of the wavefront fractions do not separate from the obstacle. As the wavefront fractions pass the top (distal) portion of the obstacle, they remain attached and therefore continue to ride along the distal contour of the obstacle, finally recombining.

This example shows the importance of separation in arrhythmogenesis. Since the wavefront fractions could never detach (separate) from the object that fractionated them, it was always clear that they would rejoin on the distal contour of the object. Since they could therefore never take independent trajectories, the wavefront was, in a sense, never discontinuous, and therefore the obstacle was not arrhythmogenic. Because of this, fixed, unexcitable obstacles such as scars are not usually arrhythmogenic.

Figure 4.4 shows the same interaction, but in this case, the excitability of the tissue is reduced so that the plane wave propagation speed is roughly 60% of normal, a level of excitability that would occur early in the relative refractory period (this configuration was studied using a reaction-diffusion PDE model in [66]). In panel (a), the wavefront has fractionated, as in figure 4-3, but it is already clear that the behavior is different. Instead of the end of the left fraction turning sharply right to ride along the distal contour of the obstacle, it has detached and continues to travel upward. This is because, in this reduced-excitability medium, the EXCITING elements are unable to source sufficient current to sustain propagation around the sharp corner of the obstacle. It is important to note, however, that the left wavefront fraction is beginning to curve into the "shadow" region of unexcited tissue distal to the obstacle.

In panel (b) of figure 4.4, the initial excitation wavefront has reached the top of the figure. Both ends of the fractions have detached from the obstacle and have *slowly* curved into the shadow region. This slow transverse propagation, and the resulting

Figure 4-4 Sequential images illustrating the collision of a depolarization wavefront with a fixed asymmetric obstacle in a medium of uniformly reduced excitability. In panel (a), one of the two wavefront fragments has just passed the left side of the obstacle while the other wavefront fragment is still riding along the right side (the obstacle is the same as figure 4-3). The left wavefront fragment curves in, but separates from the obstacle and leaves a large region of unexcited tissue (a "shadow" due to the obstacle). In panel (b), the wavefront fragments each curve into the "shadow" region, changing direction. Because the obstacle is of greater vertical extent on the right, the wave fragments collide asymmetrically, producing a transversely propagating wavelet *plus* a continuous front with a cusp as in figure 4-3. Panels (c) and (d) show two snapshots of the evolution of the wavelet into a self-sustained reentrant pattern.



elliptical contours of the shadow region, are a consequence of the 3:1 propagation speed anisotropy in this medium. This is an important factor in this mechanism of arrhythmogenesis, as we will see below. Because of the gradual curvature of the free ends of the fractions they are now both propagating transversely. Since the object was asymmetric, the left wavefront fraction began to curve before the right wavefront fraction and the fractions therefore meet, and annihilate, off-center. This leaves a free, transversely propagating wavelet, which is the portion not annihilated in the off-center collision.

In panel (c), the free wavelet has propagated transversely across the shadow region and past the obstacle. This is why the tissue anisotropy was important. The slow propagation speed in the transverse direction allowed the tissue away from the obstacle to recover excitability, or equivalently, shortened the action potential wavelength of the free wavelet. In panel (d) the wavelet is completely free of the original obstacle and has formed into a rotor which is self-sustaining.

4.5 Altering Effective Tissue Excitability

We have now seen arrhythmogenesis by a fixed obstacle (scar) in tissue of reduced excitability, because the maximum front curvature that the tissue could support was limited by the inability of the tissue at the end of a wavefront fraction to source current. Nonetheless, scars are not normally considered arrhythmogenic because, in myocardial tissue of normal excitability, fractionated wavefronts will "stick" to scars and recombine, rather than forming free rotors (Figure 4-3). This motivation is often given for the "unidirectional block" hypothesis, in which separation occurs through the disappearance of the fractionating region.

I will describe three conditions, however, which can lower the *effective* excitability of myocardial tissue and thus predispose to the type separation shown in figure 4.4:

relative refractoriness, which is a transient depression in tissue excitability at the beginning of recovery; diffuse myocardial necrosis, in which a lack of viable cells (or coupling gap junctions) decreases the average ability of the tissue to source excitation current; and small-scale dispersion of refractoriness, in which a temporary lack of excitable cells creates the same effect.

4.5.1 Relative Refractoriness

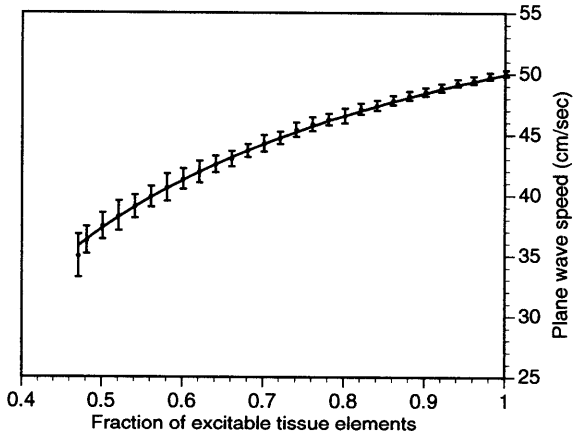
A wavefront propagating in the refractory wake of a previous wave will encounter tissue of reduced excitability. Even in healthy, homogeneous tissue, this will slow the propagation of the excitation wavefront ("decremental conduction"). In conjunction with restitution, this is believed to underlie the oscillatory behavior observed in sufficiently small rings of myocardial tissue [20, 29], and, under special circumstances, unidirectional block [32]. An excitation wavefront traveling in partially-recovered tissue will also separate from the obstacle if the recovery level is low enough (early in the recovery phase), as seen in figure 4.4.

If an ectopic beat follows a normal beat, however, it is unlikely to propagate very far in the refractory wake. This is because the waveback of a normally propagated wave moves at the same speed as the wavefront. Since a normal (non-premature) beat propagates in fully-recovered tissue, it travels faster than a wavefront traveling in partially-recovered tissue. This means that the partially-excitable waveback will "run away" from the next wavefront propagating in the refractory wake, before sustained reentry is established. Relative refractoriness is therefore probably not the principle factor leading to arrhythmias caused by single PVCs, though it would play a more significant role in sequences of several high-frequency beats (burst pacing).

4.5.2 Diffuse Necrosis

Consider a process in which diffuse myocardial damage lowers the *average* viability of the tissue. Such a process could be diffuse collagen deposition, occurring in certain types of myocarditis or in the transition regions surrounding a healed infarct; deposition of amaloid; or a disease that causes the death of small ($< 1\,\mathrm{mm}$) patches of tissue. Under these conditions, the total amount of excitation current sourced by the excitation wavefront would be reduced by the fraction of non-viable tissue.

Figure 4-5 Measured plane wave speed *vs.* fraction of excitable tissue elements. The points are the average wavefront speed obtained over 50 measurements of the spatial front shift after each time step, Δt. The error bars represent one standard deviation. The curve is the theoretical prediction for the model [26]. In all cases, the wavefront remained contiguous (did not *fractionate*). Block occurred at a fraction of 0.47.



To quantify this effect, I created a "micro-infarction" model, where a fixed fraction of the model elements were permanently unexcitable, and then computed the speed of plane waves in this model. Figure 4-5, shows the average plane wave speed measured as a function of the fraction of excitable elements in an isotropic medium. When this fraction is 1.0, the plane wave speed is the same as in a uniform, fully-recovered medium. When only 50% of the elements on the front are excitable, and thus able to source to their neighbors, the speed is reduced by almost 30%. In all cases shown in figure 4-5, the wavefront remained intact (did not fractionate). In other words, macro-

scopically, the wavefront appeared to be a normal plane wave traveling at reduced speed. This effective reduction in excitability can lead to the same phenomenon of wavefront-obstacle separation seen in a medium with uniformly reduced excitability.

Figure 4-6 Sequential frames demonstrating Wavefront-obstacle separation in isotropic tissue with a fixed fraction of unexcitable elements. Note that the small-scale stochastic variation in the distribution of unexcitable elements generates macroscopic asymmetry. Also note the continuous appearance of the individual wavefront fragments.

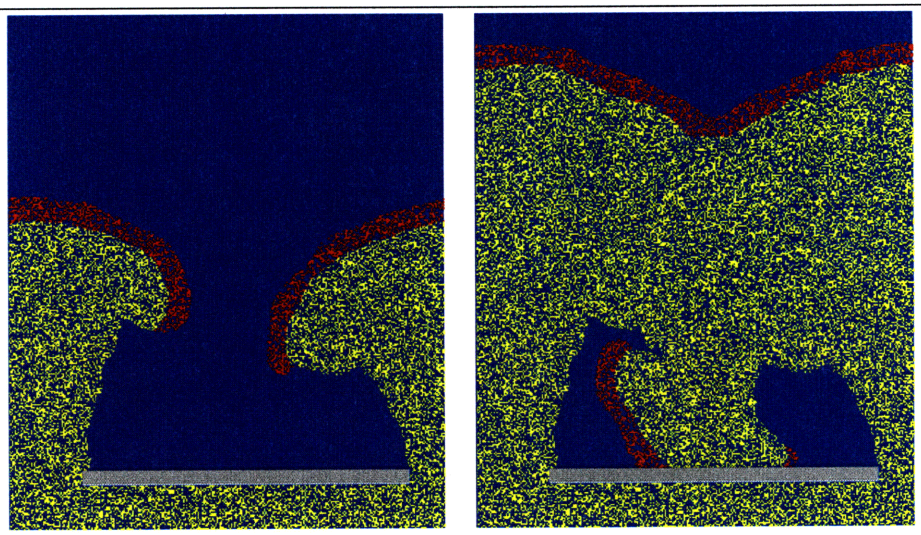


Figure 4-6 shows the interaction of a plane wave with a rectangular obstacle in an isotropic medium with a fixed fraction of unexcitable elements. Here again the wavefronts are continuous but, because of the reduction in the effective excitability, the EXCITING elements on the two wavefront fragments are unable to source sufficient current for the wavefront to turn the corners of the obstacle, resulting in separation similar to that in figure 4.4(a).

4.5.3 Small-scale Dispersion of Refractoriness

It is well-known that dispersion of refractoriness can significantly alter wave propagation in cardiac tissue [7,37,45,51,59,62]. In tissue with dispersion of refractoriness, there is a significant window of time following the passage of a wave during which

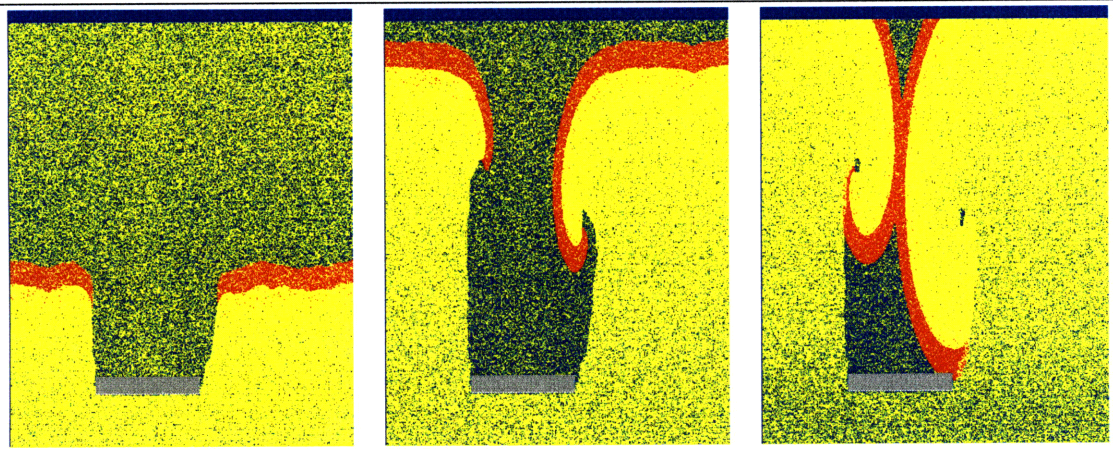
some fraction of the tissue has recovered partially or fully while the remainder is unexcitable. The fraction of unexcitable tissue seen by a wave impinging on the refractory wake of another will depend on the local recovery time (phase), T, and on the specific distribution of the refractory periods in the medium [64].

Consider a dispersion of refractoriness in which the refractory periods are correlated spatially on the scale of a typical cardiac myocyte ($\approx 100\,\mu$), *i.e.*, on a scale significantly smaller than the width of the transition region of the wavefront (≈ 1 mm). To understand wave propagation under these conditions, consider a plane wave propagating in the refractory wake of a previous plane wave. As the front propagates, a certain fraction of the elements along the front edge cannot be excited by the approaching front. These elements will be thus unable to source current to their neighbors as the wave propagates forward, which is, at that point in time, exactly the same situation as in the diffuse necrosis model described above. Therefore, the average depolarizing current sourced by the wavefront is decreased relative to the case of fully-recovered tissue. The reduction of sources on the front produces the same effect as, for example, a Na-channel blocker, i.e. it results in slower propagation. Alternatively, the decrease of the source strength can be viewed as having the same effect as increasing the strength required to excite an element (its threshold). The effect of small-scale dispersion during early recovery is thus equivalent to an effective increase in excitation threshold, which reduces the plane wave propagation speed relative to that in a medium with no dispersion.

4.6 Rotor Formation and Arrhythmogenesis

Having established wavefront-obstacle separation in the case where the fraction of excitable elements is a static property of the medium, I now demonstrate this phenomenon in tissue where the fraction of excitable elements varies dynamically due to

Figure 4-7 Interaction of a plane wave with a fixed obstacle in normally-excitable myocardium with diffuse dispersion of refractory periods. The plane wave shown was initiated 440 msec after passage of a previous plane wave (an ectopic beat). The partially-recovered tissue appears "dark-yellow" because of the small-scale mix of REFRACTORY (yellow) and RESTING (blue) elements. In panel (a), the wavefront fragments separate from the obstacle, leaving a "shadow" region in which the tissue continues to recover from the previous wave. In panel (b), the wavefront fragments have curved into the shadow region similar to panel (b) of figure 4.4. In this case the right wavefront fragment has propagated further into the shadow region, despite the fact that the obstacle is symmetric. This is because the random dispersion breaks the symmetry. In panel (c) the spiral arms of the wavefront fragments have collided and produce a transversely-propagating wavelet. This particular sequence generated a self-sustaining reentrant pattern.



(a) (b) (c)

the spatial dispersion of refractoriness in the medium. Figure 4-7 shows wavefront-obstacle separation for a wavefront propagating through the partially-refractory wake of a previous wave in such a medium. The wavefront shown was initiated 440 msec after an initial plane wave had passed through the given location. The medium had a mean refractory period of 370 msec, and a σ of 107 msec. As seen in the figure, even though the obstacle is symmetric the stochastic properties of the dispersion provide the asymmetry between the left and right wavefront fragments. Because the separate fragments meet off-center, they form a transversely-propagating wavelet, as in figure 4.4(b). This scenario results in a self-sustained pattern.

4.7 Discussion

In heart tissue, the *fractionation* of action-potential wavefronts by unexcitable obstacles which are surrounded by otherwise normally-excitable tissue is not a sufficient condition for the generation of reentrant arrhythmias in the tissue. This is because the fragments of the wavefront remain attached to the obstacle, circumnavigate it, and the rejoin on the distal side. In tissue with dispersion of refractoriness, but otherwise normal excitability, fractionation of a wavefront by islands of refractory tissue alone cannot explain the development of reentry. Arrhythmogenesis in such cases is usually explained by the scenario termed "unidirectional block." In this scenario, slowly-recovering regions of tissue are refractory to an impinging wavefront, but recover before the wavefront passes completely, allowing retrograde, but not antegrade, conduction. This is most easily explained if regions of slowly-recovering tissue are quite large compared to the size of the oncoming wavefront (*i. e.* of order 1 cm), which implies that the recovery time of the tissue need be essentially constant over this scale (large correlation length), or that smaller regions of recovering tissue fortuitously join to form large ones.

I have shown that, in tissue with sufficiently small-correlation-length dispersion of refractoriness, neither unidirectional block nor wavefront fractionation by recovering tissue is necessary for the formation of reentrant arrhythmia. This suggests that correlation of refractory periods over macroscopic spatial length scales is not a necessary precondition for arrhythmogenesis. Rather, dispersion of refractoriness at any length scale, even cellular length scales, in the presence of a fixed scar, may be sufficient to cause rotor formation, and thus reentry. The basic physical mechanism was first demonstrated in PDE models by Starobin *et.al.* [66] and Pertsov*et.al.* [56], who showed separation and rotor formation in the presence of a single, fixed obstacle in an otherwise uniform medium of low excitability. I have demonstrated a similar mechanism in a finite element model of heart tissue, where the nominal excitability was

normal, but where the dispersion of refractoriness on a small spatial scale results in a reduction of *effective* excitability for wavefronts propagating in the partially-recovered wake of a previous excitation. In our model the requirement for the asymmetry of the obstacle is absent, because the stochastic variation in the distribution of refractory periods generates the needed asymmetry.

There are several possible clinical implications to this hypothesis. First, it is important to note that current extracellular electrophysiologic techniques may not be able to detect the small (cellular) scale dispersion of refractoriness addressed in this study. Such small-scale patterns of dispersion may become apparent when the action potentials of neighboring cells are directly measured using intercellular electrodes. Also of clinical significance is the fact that a normally innocuous fixed scar, such as a surgical scar or a small, healed infarct, may become arrhythmogenic once the small-scale dispersion of refractoriness exceeds a certain threshold. This suggests a possible mechanism for sudden cardiac death via the progression of disease responsible for small-scale dispersion.

Although the quantitative results obtained by this model can be considered physiologically relevant given the tissue parameters used, it will be useful to devote further study to obtaining more precise values of dispersion and tissue parameters. Such values could come from detailed experimental studies measuring action potentials of individual cells in situ. The information obtained from such studies could be used to make more detailed predictions as to the specific vulnerable window for this type of arrhythmia formation, as a function of dispersion of refractoriness, obstacle size, and timing of a premature beat.

Chapter 5

Induction of Arrhythmia and Preemptive Pacing

In this chapter, I will use the simulator and the results from chapter 4 to study the possibility of using preemptive pacing to stabilize a diseased heart against arrhythmia. This study is based on the hypothesis that a low-energy diffuse-field pacing pulse, if administered in response to a detected arrhythmogenic event such as a PVC occuring close the end of the effective refractory period, could capture enough myocardial tissue to prevent reentrant arrhythmia from developing. A similar hypothesis, using multiple electrodes rather than a uniform field, was tested experimentally in 1992 with inconclusive results [17]. I will use the simulator to study the effectiveness of the idealized, diffuse-field preemptive protocol, with the goal of determining why the experimental study failed and studying the practicality of the technique.

In section 5.1, I will describe the process of systematically inducing sustained arrhythmia in a model of the ventricle. Arrhythmia usually occurs in the context of structural cardiac disease and as the result of an unfortunately-timed electrical event. I will first describe the modifications I made to the modeled electrical substrate to incorporate the presumed effects of this structural disease. I will then describe the

stimulation protocols used to induce arrhythmias on these substrates.

In section 5.2, I will demonstrate the testing of techniques for preemptively pacing "into" a PVC to reduce the probability that the PVC will result in lethal arrhythmia. I will evaluate and compare techniques will discuss likely failure modes.

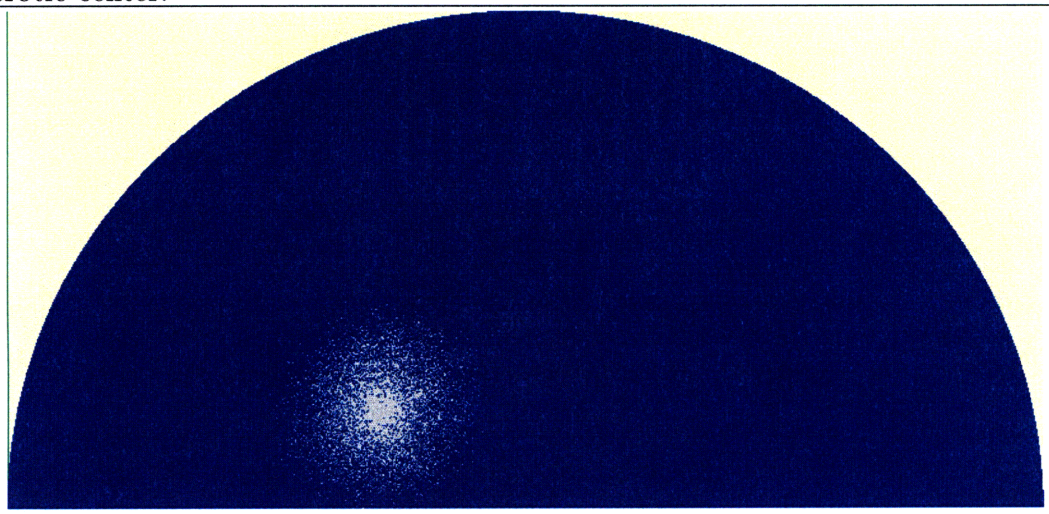
5.1 Arrhythmia Induction

The simulator would be useless in this study if it could not be made to demonstrate arrhythmia. Chapter 4 described several local conditions which can lead to arrhythmia. In each case the potentially arrhythmic local event was initiated by carefully-controlled patterns of stimuli. In the heart such control, even in the electrophysiology laboratory, is impractical. Arrhythmia will develop "accidently" when the interaction of multiple excitation waves creates the necessary local conditions in a susceptible region of tissue. Further, a large region of tissue surrounding the region of initial arrhythmia must be in a state that allows the arrhythmia to be "captured" by the remainder of the heart, and thus sustained.

Patients with structural cardiac disease that is sufficiently severe to result in an episode of sudden cardiac death often do not develop subsequent lethal arrhythmia [58]. Of those who do, the arrhythmic event may not occur for many months [58]. Clearly, a simulator that accurately represents the electrophysiology of the heart will develop arrhythmias only under very special circumstances. It is therefore significant that the model was very stable against arrhythmogenesis.

The arrhythmia induction model used is similar to arrhythmia induction protocols I observed in the electrophysiology laboratory. The exact parameters used for induction, both in terms of disease model and exact sequence of stimuli, were only found after an extensive period of trial-and-error.

Figure 5-1 The frame shows the infarction alone: the gray "speckled" region centered just above the AV annulus. The gray elements are unexcitable and the mix of gray and blue (RESTING) is the transition (partially-viable) region surrounding the completely necrotic center.



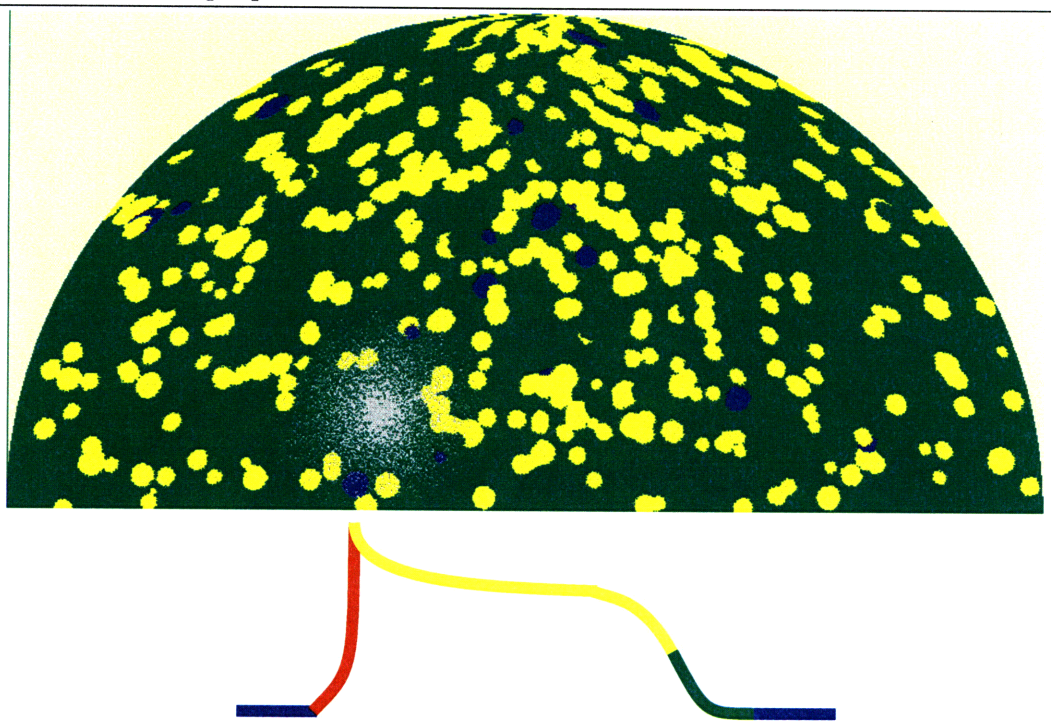
5.1.1 Disease Model

Infarction

Myocardial infarction has been found to be strongly correlated with ventricular arrhythmia [41,58]. One of the models often advanced for this correlation is the existence of "transition region" surrounding the infarction in which the tissue is only partially viable. Such a region would result in slowed conduction and increase the probably of unidirectional block in the region.

I therefore chose to incorporate a "healed infraction" model into the substrate (Figure 5-1). A circular region of partially viable tissue (as described in section. 4.5, under **Diffuse Necrosis**), was located near the tricuspid annulus. The viability of the tissue increased radially continuously from 0 (no viable tissue) at the center of the region to 1.0 (completely viable) at the edge. The size and location of the region placed the edge of the partially-viable region (the "transition region") at the tricuspid annulus. This created a "one-dimensional" path of decreased excitability between the infarction and the annulus.

Figure 5-2 Infarction and Dispersion of Refractoriness in a Diseased Substrate. This frame shows the small patches from dispersion of refractoriness. Some of the patches have abnormally *short* refractory times and thus have already recovered (blue). Most patches have abnormally long refractory times and are slow to recover (yellow). The exact distribution of disease and structure of the infarct varied between substrates, but the statistical properties were the same.



Dispersion of Refractoriness

Dispersion of refractoriness is often suggested as a precipitating factor in ventricular arrhythmia [7, 8, 37, 45, 51, 52, 59, 63]. The dispersion is often quantified by determination of the local effective refractory period using pacing electrodes [51, 52, 59], and it is therefore difficult to be sure of the exact spatial characteristics of the dispersion. Many studies have reported dispersion only as the largest difference in the refractory periods measured at a small number of sites, which contains very little detailed information about the length scale of spatial variation [37, 51, 59].

From the mechanisms of arrhythmogenesis discussed in chapter 4 it is clear that the spatial correlation length of the dispersion must be $\gtrsim 1$ mm to cause fractionation

and therefore polymorphic arrhythmia or VT. It is also clear that the extension of refractory period must be significant compared to the nominal refractory period, so there will be "islands" of refractory tissue which are surrounded by excitable tissue and cause fractionation. I therefore chose a model in which small ($\sim$ 2–4 mm) circular regions of tissue had refractory periods which were approximately 60% longer than the surrounding tissue (Figure 5-2).

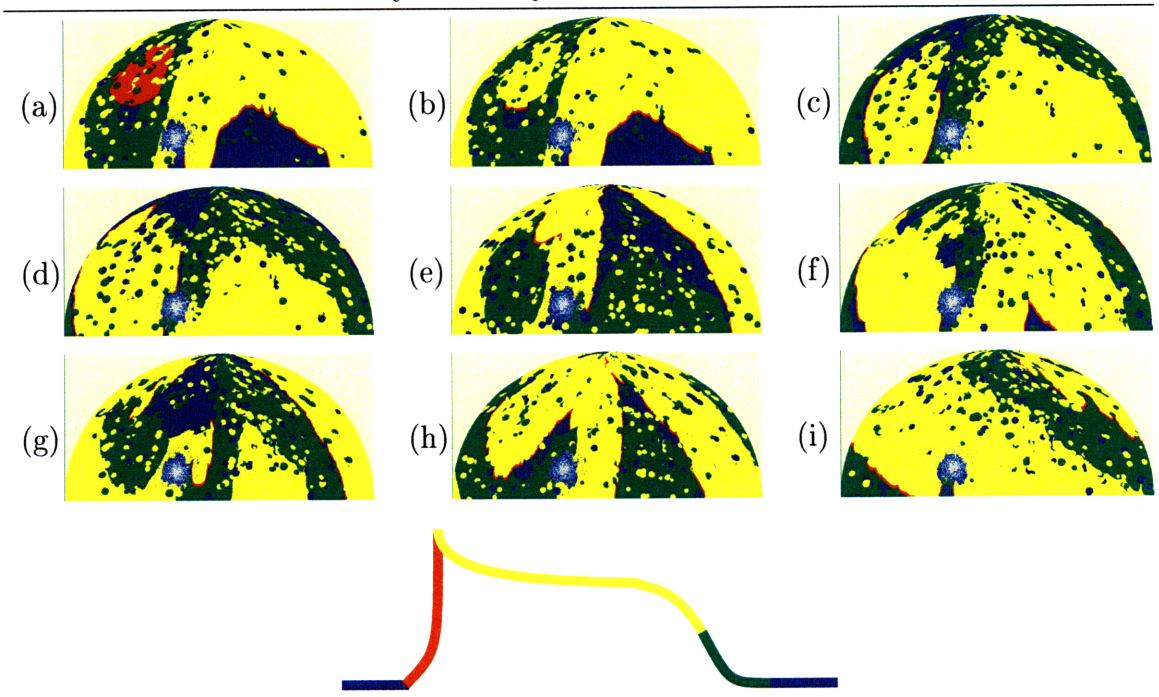
5.1.2 Induction Protocol

"PVC" Induction

This protocol was based on the assumption that a single PVC occurring in the vulnerable period of partial repolarization ("R-on-T") had a high probability of leading to sustained arrhythmia. This protocol was subsequently used for all pacing studies since it provided a single critical point that could presumably be detected by an automatic device: the final "PVC" in the sequence. All interventions were referenced in time to the occurrence ("detection") of this PVC. The protocol was started with four rapid beats of ventricular pacing (130 BPM). A location within 3 cm of the ventricular pacing site was then chosen for introduction of the PVC. The site was stimulated at a specific delay from the last pacing stimulus, and the delay was increased in 10 msec intervals until capture was detected. This was taken to be the "time" of the PVC, that is, the time of first depolarization.

The PVC was modeled as a circle of fixed radius between 1.1 cm and 1.8 cm, which was initiated 30 msec *after* the time of first excitability and therefore represents an "evolved state" of the PVC propagation. This additional delay was incorporated to allow capture of most of the circular region. The area of the circular region loosely corresponded to a DC "ventricular fibrillation threshold," since the size of a region excited by a local pacing electrode is a function of the (DC) charge delivered through

Figure 5-3 Sequential frames showing the induction of sustained arrhythmia by rapid pacing followed by a "PVC". Four ventricularly-paced beats at 130 BPM were followed by a "PVC" at the end of the effective refractory period. Panel (a) shows the state of the substrate when the PVC occurs. There is a small excitable region beyond the PVC and moderate dispersion of refractoriness evident. In panel (b) the stochastic nature of the capture is evident: only the tissue immediately *below* the PVC was sufficiently excitable to support propagation. Panel (c) shows the wavefront moving upward along the central REFRACTORY region from the uncaptured portion of the PVC. In Panel (d) there is a completely-independent wavefront fraction above the infraction which has been "cut" by the infarction below and a cluster of RELATIVE REFRACTORY tissue above. The fraction on the far left is still attached to the central REFRACTORY region. Note that the leftmost part has wrapped around to the extreme right. The two wavefront fractions are still independent in panel (e). The large fraction is penetrating the central (now RELATIVE REFRACTORY) region. In panel (f) the pattern is repeating but with smaller fractions due to the difference in recovery level of the tissue. Panel (g) shows many wavefront fractions resulting, which evolve to stable reentry with very little excitable tissue in panels (h) and (i).



the electrode during the pulse. An example of this protocol is shown in figure 5-3.

Table 5.1 shows the results of this protocol on each of the 15 substrates studied. Each substrate was constructed by creating a 2 cm radius partially-necrotic region centered within 2 cm of the tricuspid annulus (Sec. 5.1.1). This guaranteed that the partially-necrotic transition region abutted the annulus which consequently produced a "one-dimensional" slow conduction pathway. The exact location of the center varied from 1.775 to 1.9 cm below the annulus. Regions of extended refractory period were added to cover approximately 25% of the lattice. Mean refractory times in these regions were 390 msec, and the characteristic size of the regions was 4 mm.

5.1.3 Discussion

The fifteen substrates used in the induction study provided a variety of microscopic conditions while all having the same statistical properties. To the extent that electrical inhomogeneities modeled can be thought to be consistent with organic disease processes, all substrates would be at the same level of "disease." Generation of arrhythmia is a statistical process and as such depends on the stochastic nature of the substrate and it's interaction with the stimuli applied. This is reflected in the induction results. As can be seen from Table 5.1, there were substrates that were induced on every attempt, and substrates that could not be induced with any attempt. Although large spot sizes increased the induction rate, it was still impossible to predict the effects of a spot on any particular lattice.

The broad statistical variation of induction success across this set of statistically-similar substrates suggests that they will form the basis of a good test of arrhythmia prevention techniques.

Table 5.1 Success in Inducing Sustained Reentrant Arrhythmia on Each of 15 Substrates. Each substrate was statistically similar but had a slightly different infarct location and a different microscopic distribution of refractoriness. Those substrates marked with a bullet (•) developed sustained arrhythmia after pacing from a circular region with the specified radius, 30 msec after first recovery of the center of the spot.

	"PVC" Spot Radius		
Substrate	1.1 cm	1.45 cm	1.8 cm
1		•	•
2			•
3	•	•	•
4	•		•
5	•	•	
6		•	•
7		•	•
8			•
9			•
10		•	•
11			
12	•	•	•
13	•	•	•
14		•	•
15		•	•
Induction Rate	33%	67%	87%

5.2 Pacing

The pacing model is described in section 3.4. Here I will describe the specific choice of pacing considerations for these studies.

5.2.1 Uniform-field Pacing

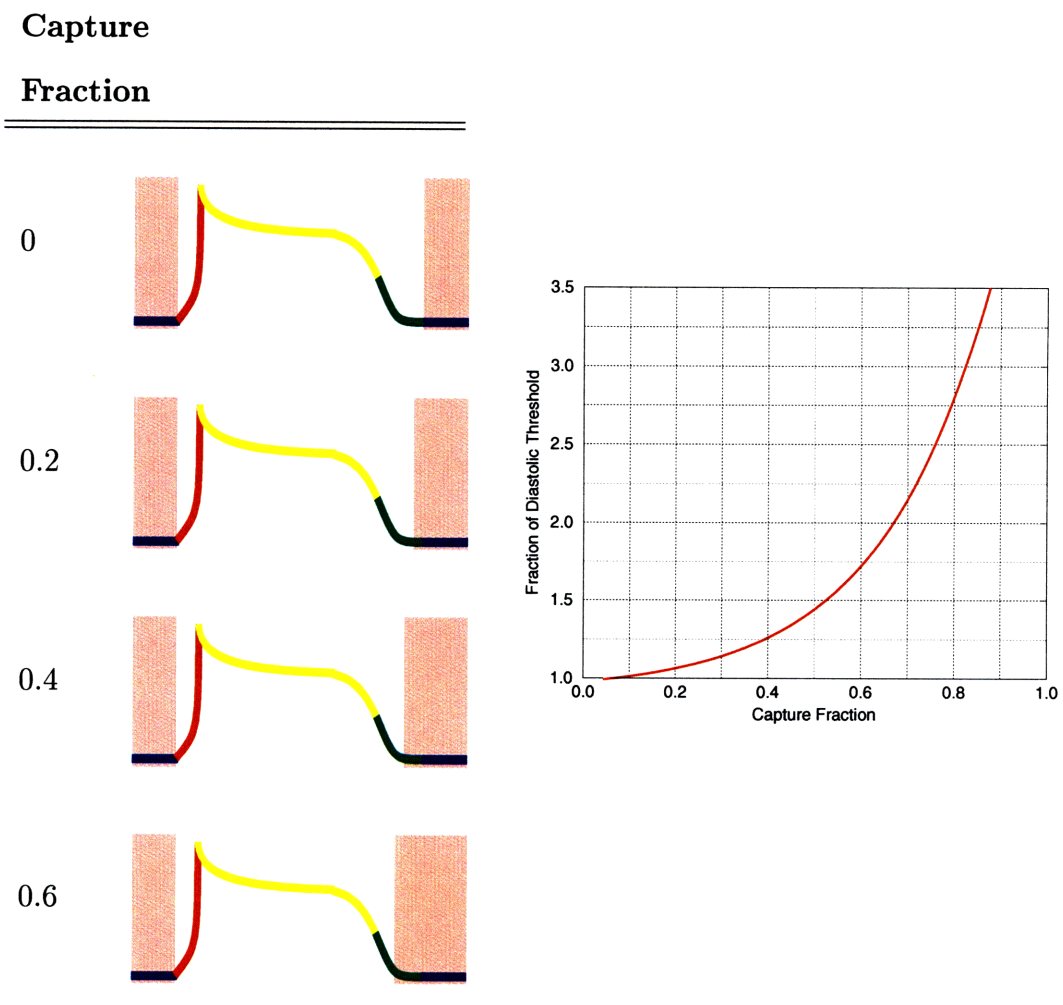
One of the principle questions in this study is whether a low-voltage pulse, applied uniformly to the ventricle, would make it less susceptible to arrhythmia. Perfect uniformity of field is obviously impractical given current technology, but as an idealization it provides a good first test of the hypothesis. A uniform field pulse was modeled as a pacing stimulus that did not distinguish elements based on location, that is, all elements in the substrate were selected for stimulation based on excitation threshold.

5.2.2 Pacing Strength

Stronger pacing pulses can capture (excite) tissue that is less recovered. As described in section 3.4, the selection of elements for excitation by a particular pacing pulse is based on the excitability of those elements. For reasonable electric fields, there is a monotonic relationship between the voltage (or current) of a pacing impulse and the recovery level of the tissue which will be captured, the *strength-interval* curve. This curve is very non-linear, reflecting the fact that small changes in recovery time can result in large changes in excitability.

Since the strength-interval curve is monotonic, pacing strength could be specified linearly in "voltage," i.e. different experiments could sample constant widths in pacing voltage. The non-linearity of the curve would result in poor sampling of tissue recovery, however. The lowest voltage range would correspond to the last 50–75% of the recovery process and the higher ranges would only include small amounts of

Figure 5-4 Definition of Capture Fraction in terms of the Cardiac Action Potential. The figures on the left show the recovery levels of elements that will be excited by the pacing pulse. All elements with recovery levels in the red highlighted regions of the curve will be excited. The graph on the right shows the "strength-interval" relation for capture fraction as the increase above diastolic threshold required to get a particular capture fraction.



additional tissue. For this region, the choice of pacing strengths was based directly on the level of tissue recovery, rather than directly on the strength-interval curve. An illustration of this definition is given in figure 5-4. A calculated strength-interval curve is plotted against capture fraction in the same figure.

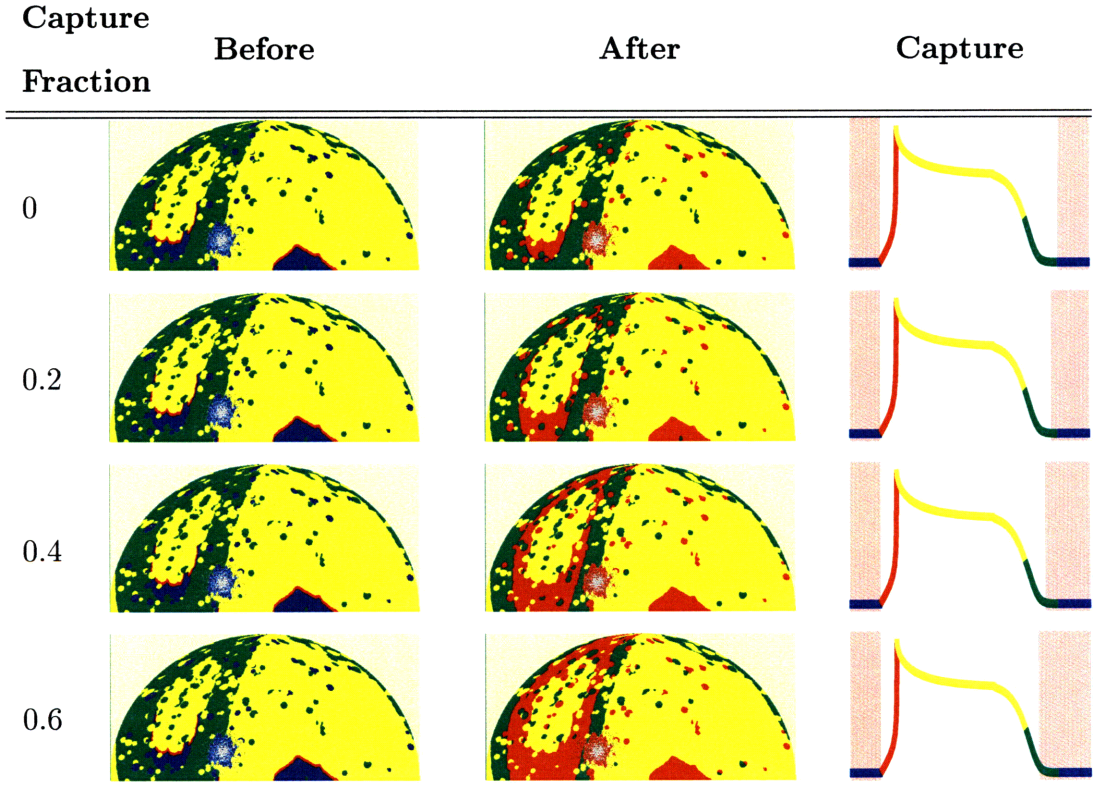
The simulator models recovery as a continuous process, starting when an element switches from the REFRACTORY to the RELATIVE REFRACTORY state and continuing until the tissue is fully-recovered and switches the the RESTING state (Sec. 3.2.1). Although the action potential duration, and thus the duration of the REFRACTORY state varies spatially and dynamically, the duration of the RELATIVE REFRACTORY state—the time from when an element first regains excitability to the time when it regains full excitability—is fixed.

Since each element regains excitability at the same rate, choosing to sample fixed widths in "recovery space" implies that each increase in pacing strength will capture roughly the same additional amount of tissue, depending on the fraction of elements at each stage of recovery (see figure 5-5). A pacing strength of 0 captures only fully-recovered tissue (RESTING elements). A pacing strength of 0.2 captures all tissue captured at 0 strength *plus* tissue that has completed $(1 - 0.2) = 80\%$ of the RELATIVE REFRACTORY phase. A pacing strength of 0.4 captures all tissue at least 60% recovered. Using this experimental protocol allows the study of the capture of progressively increasing increasing amounts of partially-excitable tissue, and thus allow a thorough study of the consequence of pacing into increasingly unexcitable tissue.

5.2.3 Pacing Protocols

The theoretical advantage of uniform-field pacing would be the simultaneous capture of all excitable tissue in the heart. This is similar to the principle of defibrillation, where all tissue is captured and therefore rendered refractory for some period of time.

Figure 5-5 Examples of the effects of uniform-field pacing pulses at various strengths (capture fractions). The rows show the effects of uniform-field pacing pulses at capture fractions ranging from 0 to 0.6. Each pulse is at the same point in the simulation. The second column (first picture) is the state of the simulation immediately before the pulse. The third column is the effect of the pulse. The fourth column is a illustration of the recovery levels of elements that will be excited by the pacing pulse. All elements with recovery levels in the red highlighted regions of the curve will be excited. As can be seen, each increase in capture fraction increases the amount of tissue captured and decreases the amount of RELATIVE REFRACTORY (green) tissue remaining after the pulse.



The difference in this case, however, is that not *all* tissue would be captured, since that requires substantial energy and potentially involves substantial discomfort to the patient. Rather, uniform-field pacing pulses at relatively small energy would be used to capture as much tissue as possible.

ICDs give large-field shocks to the heart as soon as possible after the detection of life-threatening tachyarrhythmia. The defibrillation threshold is defined as the minimum shock that must be given under these circumstances to terminate the arrhythmia. The purpose of this study is to explore the possible use of lower-energy shocks. This obviously can only be successful if the shock timing is modified relative to the ICD case.

I therefore tested the following hypothesis:

> If a dangerous (potentially arrhythmogenic) PVC could be detected, a uniform-field pulse, or train of pulses, immediately after detection of the PVC, would excite enough tissue to prevent the PVC from propagating into a sustained reentrant arrhythmia.

An important aspect of this hypothesis is the definition of *dangerous PVC*. In this study, a *dangerous PVC* was defined in terms of the induction protocol (Sec.5.1.2). Since the purpose of the protocol was to generate sustained arrhythmia on as many substrates as possible, I reformulated the hypothesis as follows:

> A uniform-field pacing pulse or train of pulses, delivered after the last PVC in the induction protocol described in section 5.1.2, will significantly lower the rate of sustained arrhythmia.

All pacing studies had the same structure. The same 15 substrates used in the original induction studies were used for each pacing study. A control case (no intervention) was run to determine if the protocol would generate sustained arrhythmia. Each potential pacing intervention was then run independently and compared to the

control. If sustained arrhythmia occurred in the control case but did not occur after the intervention, that intervention was counted as a success ("Successful Prevention"). If sustained arrhythmia *did not* occur in the control case but did occur after the intervention, that intervention was counted as a failure ("Spurious Induction"). The cases where the intervention did not change the outcome were not counted.

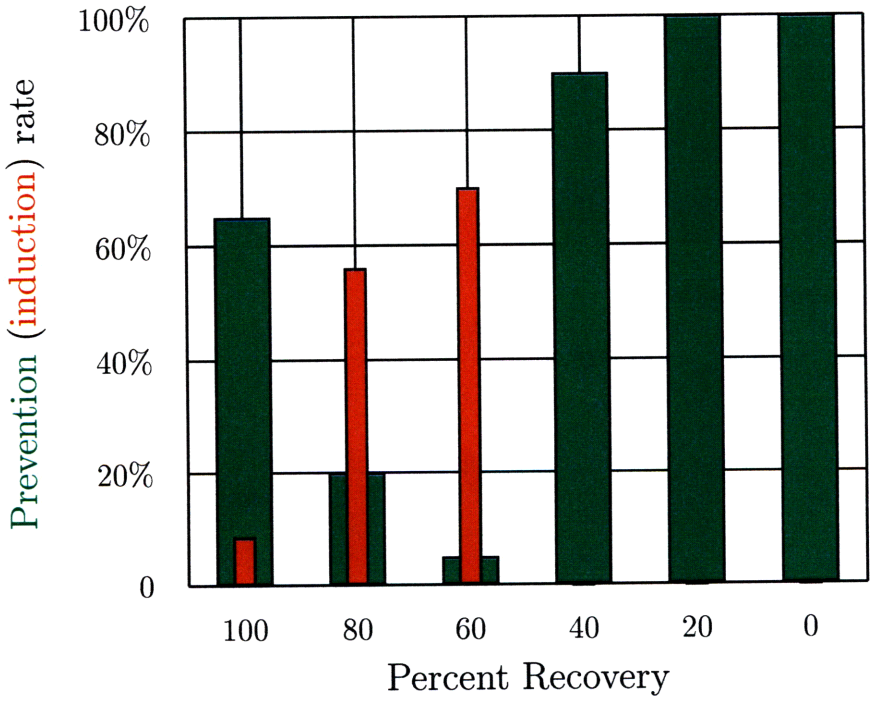
Protocol I: Single Field Impulse Immediately After Detection of PVC

This protocol attempted prevention of sustained arrhythmia with a single uniform-field pacing pulse, delivered at a fixed interval, 80 msec after the final PVC in the induction protocol. This was attempted for 6 independent field strengths. The lowest field strength, which is designated here as a "capture fraction" of 0, caused the excitation of only elements that were fully-recovered (RESTING). Thus the capture fraction of 0 refers to the fact that *no* RELATIVE REFRACTORY elements were captured by the pulse. A capture fraction of 0.6 means a RELATIVE REFRACTORY element would be captured if it was at least 40% $(1 - 0.6)$ completed with recovery. This would generally correspond to approximately 60% of the elements in the RELATIVE REFRACTORY state.

As the strength of the pacing pulse was increased, more potentially-excitable tissue was captured and therefore unavailable for propagation of a sustained arrhythmia. It would seem therefore that a high capture fraction (capturing all or most of the available excitable tissue) would be very successful in preventing sustained arrhythmia, especially since this case represents the defibrillation endpoint. By this reasoning, it might also seem that as power increased, the success at preventing sustained arrhythmia would increase, and similarly the rate of accidently initiating sustained arrhythmia would decrease.

Figure 5-6 shows the success and failure rates for the single pulse protocol as a function of pacing strength. As expected, the high-strength (defibrillation-level)

Figure 5-6 Preemption *vs.* Induction Rate for Single Field Pulse at Various Strengths. A single PVC was used to attempt arrhythmia induction, followed 80 msec later by a single uniform-field pulse. The green bars represent cases where the pulse prevented sustained arrhythmia that occurred in the control (no uniform-field pulse) case ("Successful Prevention"). The red bars represent cases where the pulse generated sustained arrhythmia that did not occur in the control case ("Spurious Induction").
The "Capture Fraction" shown on the x-axis shows the level of recovery captured by the pulse (Sec. 5.2.2, and figure 5-4).: a capture fraction of 0.4 means that all elements having only 40% or less of the recovery process *remaining* (having recovered 60% or more excitability) were excited by the pulse. A capture fraction of 0 means that only fully-recovered elements were excited.



fields are completely successful. What is surprising, however, is that the very-low-strength fields (especially those capturing *only* fully-recovered tissue) appear to be much more successful that the intermediate strength fields which capture some, but not all, relatively refractory tissue). In particular, the pulse which captures 40% of the RELATIVE REFRACTORY elements (capture fraction of 0.4) clearly does much more harm than good.

Understanding of this result requires a consideration of the detailed local mechanisms involved in arrhythmogenesis, particularly the interaction of excitation wavefronts and obstacles (Sec.4.2). A pulse with a capture fraction of 0.4 will capture, and therefore cause excitation wavefront propagation, in low-excitability tissue. As explained in section 4.5, wavefront-obstacle separation, a necessary precondition to reentrant arrhythmia, occurs much more easily in this tissue. Also, there is a greater likelihood of completely unexcitable tissue near the wavefront. A pulse with a capture fraction of 0, by comparison, will only cause propagation in normally-excitable tissue. When the resulting excitation wavefront encounters obstacles in this tissue, it will "stick" to the obstacles rather than separate (as in Figure 4-3) and therefore any obstacles remaining will not be arrhythmogenic. It is therefore understandable that medium energy pulses would be arrhythmogenic compared to low-energy pulses (Figure 5-7).

Protocol II: Multiple Field Impulses Immediately After Detection of PVC

The results from Protocol I suggest that low-energy pulses and high-energy pulses are the most likely to prevent arrhythmia. Intermediate-energy pulses are often arrhythmogenic. Since low-energy pulses would be most practical and most tolerable in patients, it is interesting to examine the low-energy case. In Protocol II, I extended the investigation from a single pacing pulse to a train of pacing pulses.

It is first useful to consider the endpoints of the study. If a single is unsuccessful

Figure 5-7 An illustration of the mechanisms responsible for prevention and induction of arrhythmia in uniform-field pacing. The top frame is the state immediately prior to "preemptive" pacing. The left frame on each row is the uniform-field pacing pulse. Capture fractions of 0.0 and 0.6 are antiarrhythmic. The top row shows the effect of a capture fraction of 0 (exactly diastolic threshold). The propagating wavefronts are smooth and continuous—there is no partially-excitable tissue to break them, and when they are fractionated the heal quickly (see figure 4-3). At 0.6, most of the partially-excitable tissue is also captured, so a propagating wave is more likely to block than to fractionate. The result is very little activity which dies quickly. A capture fraction of 0.4 is the worst of both worlds. Not all the partially-excitable tissue is captured, so it is available for propagation of the resulting excitation waves. Propagation in this tissue leads to several detached wavefront fragments, which continue to pursue independent courses (see for example, figure 4.4).

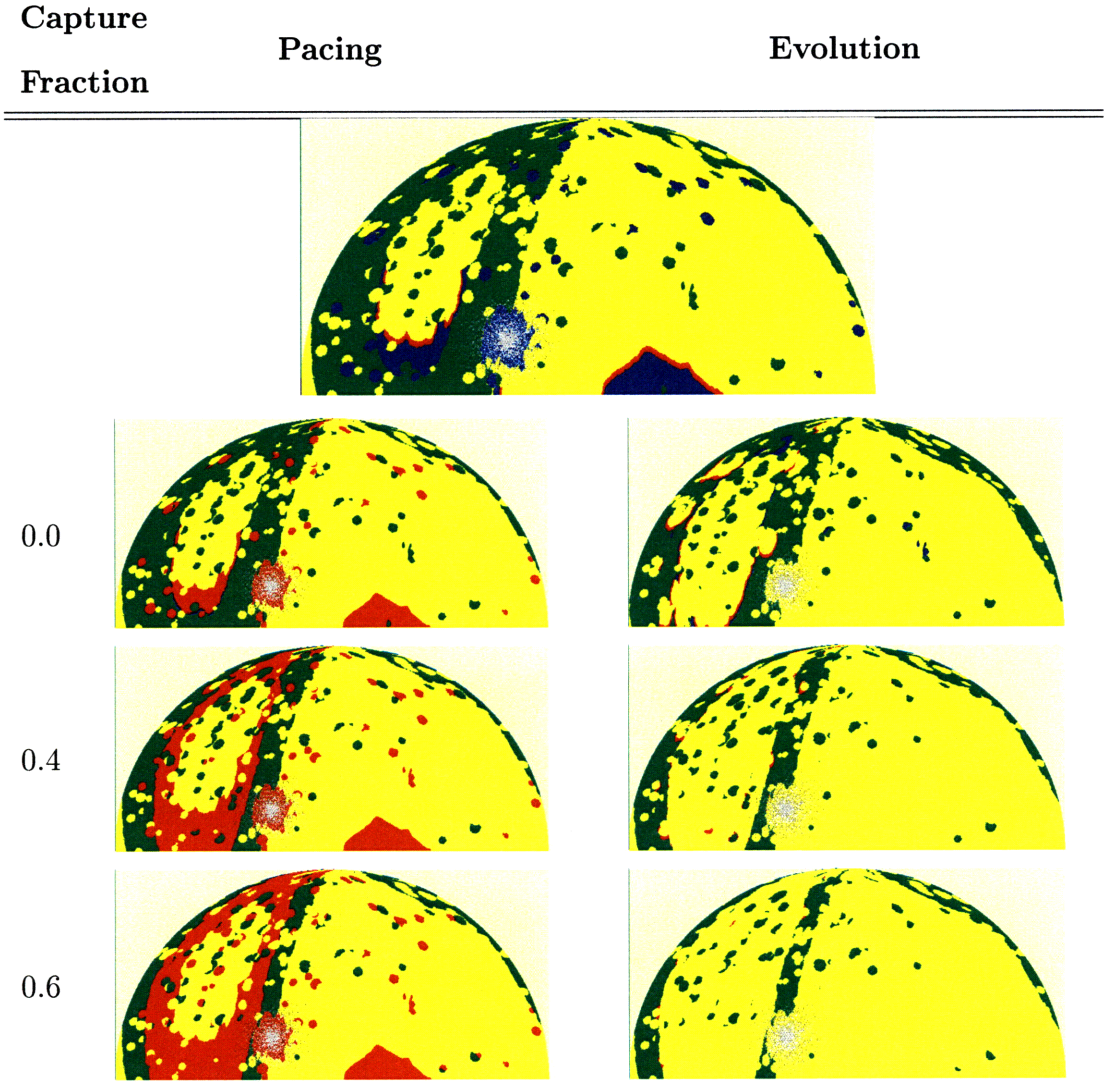
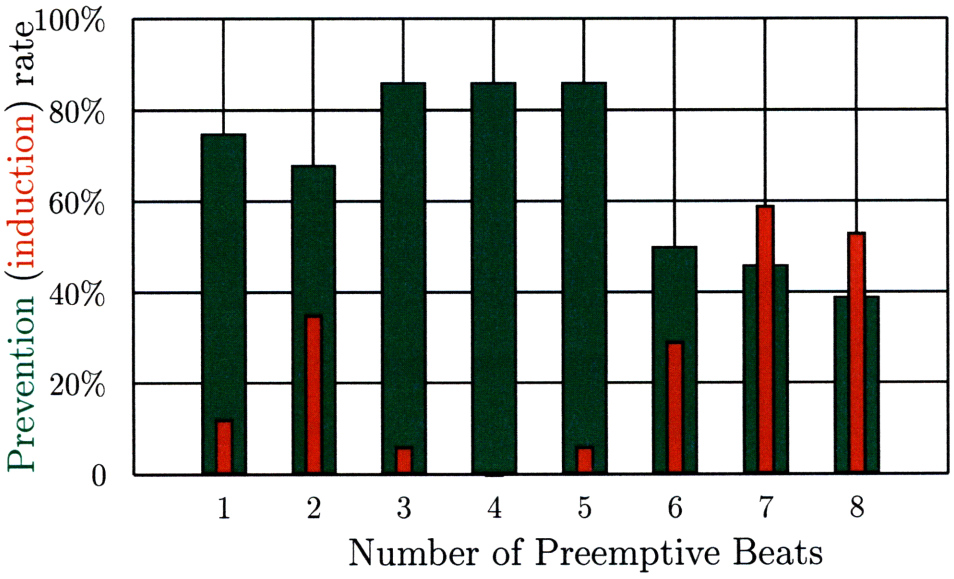


Figure 5-8 Preemption *vs.* Induction Rate for Multiple Field Pulses. A single PVC was used to attempt arrhythmia induction, the same protocol as figure 5-6. A sequence of uniform-field pacing pulses was initiated 80 msec later with a period of 40 msec. The green and red bars have the same meaning as in figure 5-6. In all cases, the capture fraction was 0, meaning that only fully-recovered tissue was excited by the pulse. The x-axis shows the number of pacing pulses in the sequence.



in preventing the arrhythmia because it failed to capture sufficient tissue, then a second pulse, delivered shortly after the first, might improve success by capturing additional tissue *without* incurring the problem of capturing partially-excitable tissue as happened with the intermediate-energy pulses in Protocol I. However, a long train of rapid pulses is similar to protocols used to *induce* tachycardia and therefore one might expect high failure rates for the extreme case also of many additional pulses.

The purpose of Protocol II was to determine the optimal length of a train of pulses for preventing sustained arrhythmia. Figure 5-8, shows the effect of a train of several, low-energy, uniform-field pacing pulses. The train was initiated 80 msec after the "time" of the PVC, with a period of 40 msec. It was applied to the same set of induction protocols used in the previous studies.

As can be seen from the figure, successful prevention of arrhythmia increases

for trains of up to five beats (there is a small decrease from 1 to 2 beats). More importantly, however, the spurious induction rate declines to 0 for 4 beats and is very small for 3 or 5 beats. This suggests that there would be a significant advantages to short trains of preemptive low-energy pacing pulses. For a long train, as expected, the successful prevention rate decreases and the spurious rate increases significantly. This is consistent with the expectation that a long train would induce arrhythmia and the decrease in the prevention rate is due to *re-induction* of the arrhythmia.

Protocol III: Single Field Impulse at Variable Delay After Detection of PVC

Protocol III was another modification of Protocol I. Protocol II was designed to determine if increasing the number of preemptive pulses would improve income. Protocol III was designed to determine the optimum delay to a preemptive pulse. The results of single uniform-field pacing pulses occuring from 0.1 sec to 0.5 sec after a PVC are shown in figure 5-9. The success rate drops sharply and the spurious induction rate increase sharply for delays greater than 200 msec, confirming the presumption the this technique is most effective when the pacing pulse is initiated before the PVC has a chance to evolve.

Protocol IV: Single Field Impulse After Detection of Arrhythmia

Spurious induction of arrhythmia in the preceding protocols is a major concern. It is therefore interesting to determine the effectiveness of uniform-field pacing technique in cases where the arrhythmia has already been established. Though this eliminates the ability to preemptively capture tissue, it also substantially reduces the problem of causing arrhythmias where the PVC itself might have been benign.

The results of pacing with a uniform-field pulse at diastolic threshold (capture fraction of 0) into an established arrhythmia are shown in figure 5-10. As can be

Figure 5-9 Preemption *vs.* Induction Rate for A Single, Uniform-field Pacing Pulse at Various Delays. A single PVC was used to attempt arrhythmia induction, the same protocol as figure 5-6, followed by a uniform-field pacing pulse. The x-axis is the time between the "PVC" and the pacing pulse. The green and red bars have the same meaning as in figure 5-6. In all cases, the capture fraction was 0, meaning that only fully-recovered tissue was excited by the pulse. The x-axis shows the number of pacing pulses in the sequence.

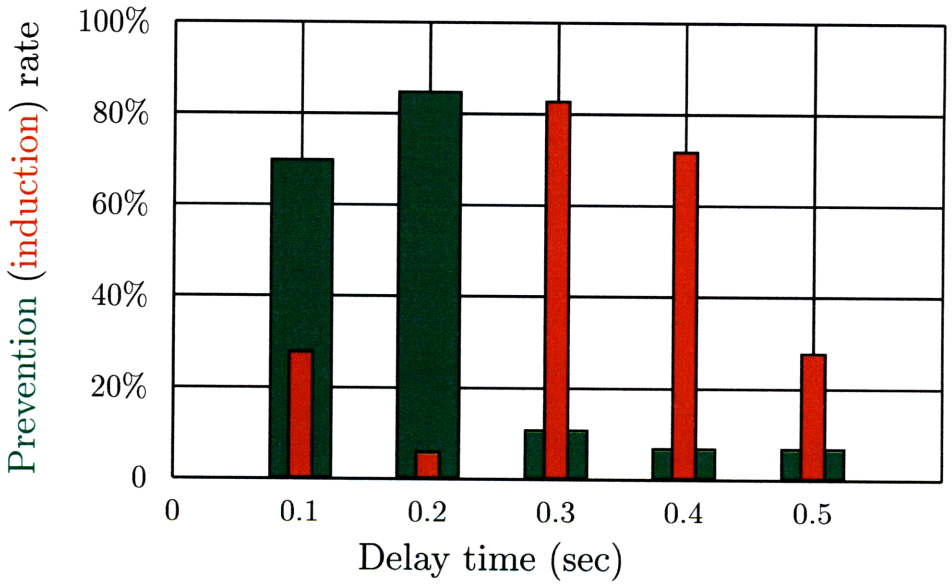
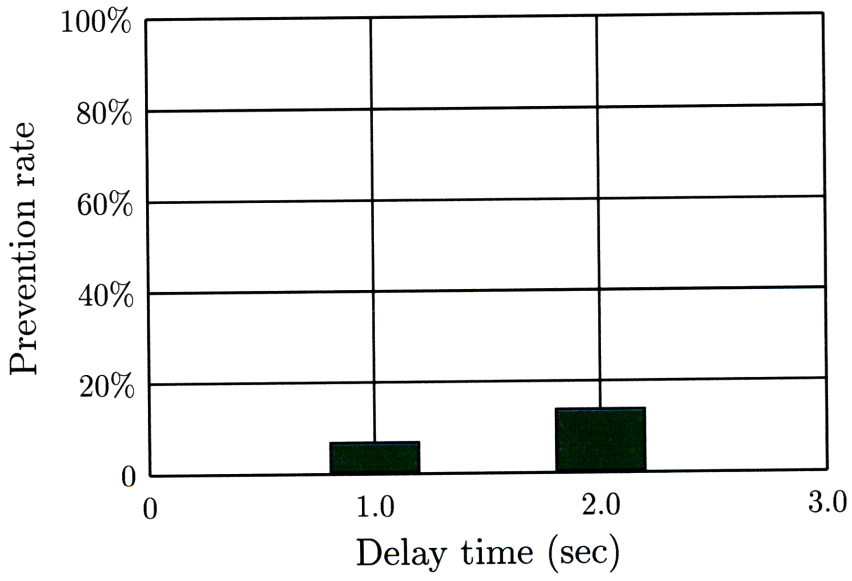


Figure 5-10 Preemption Rate for Single, Uniform-field Pacing Pulse Initiated After the Development of Arrhythmia. A single PVC was used to attempt arrhythmia induction, the same protocol as figure 5-6. For cases where sustained arrhythmia persisted for at least 1 second, a single, uniform-field pacing pulse was attempted either 1 second or 2 seconds after the PVC. The green bars show the percentage of cases where the effect of the pulse was to terminate the arrhythmia. The capture fraction of the pulse was 0.0, meaning that only fully-recovered tissue was captured.



seen, the success rates of this technique are approximately 10%, which is similar to the worst-case success rates in any of the previous protocols. This confirms the presumption that low-energy pacing would be most effectively used preemptively.

5.2.4 Discussion

Defibrillation works by applying a high-strength electric field which captures (renders unexcitable) *all* (or at least most) of the tissue in the heart, effectively blocking propagation of excitation wavefronts for hundreds of milliseconds. The disadvantages of defibrillation are energy cost (of critical importance in battery-operated implantable devices) and patient discomfort from large shocks. The hypothesis I wished to explore was that a low-energy, uniform-field pulse, applied to the entire heart would capture *sufficient* tissue to block propagation of reentrant arrhythmia. For a completely uniform field, this is born out by the simulation results, provided the field only captures fully-excitable tissue. Simulation results indicate that prevention rate for this technique is improved by using a short, rapid train of beats, and the spurious inductions are eliminated, though a sample size of 45 is obviously too small to draw strong conclusion from that fact.

For slightly higher-strength uniform fields, the technique fails for a combination of two reasons. First, the resulting wavefronts propagate in partially-excitable tissue, which leads to separation from existing electrical inhomogeneities and is therefore arrhythmogenic (Sec. 4.4). Second, wavefronts propagating in partially-excitable tissue are more likely to block, and the stochastic nature of this block leads to a high incidence of unidirectional block. Thus, intermediate strength fields have a high rate of spurious induction.

Obviously, the cases of spurious induction are highly-significant, since failures represent cases of arrhythmogenesis by the pacemaker itself. It is worth noting that the rate of sustained arrhythmia for this protocol on these substrates was 62%, so given

a very accurate algorithm for detection of "dangerous PVCs," even an outcome with occasional spurious induction might be considered advantageous against a high rate of successful prevention. It is likely, however, that the identification of "dangerous" PVCs would be difficult, and therefore any spurious induction due to a therapeutic intervention would be unacceptable.

The most significant conclusion, therefore, is that intermediate-strength fields are very arrhythmogenic. Since any available electrode configuration will produce field inhomogeneity, and since from figure 5-4 it is clear that there is not much difference in strength between antiarrhythmic and *pro*arrhythmic fields, there will always be regions of the heart where the field will be stronger than intended, and therefore arrhythmogenic. This is consistent with experimental outcomes for multiple-site pacing. The model study has now provided an explanation of that outcome. Despite the hope that the model study would also provide a solution, it appears that the problem is fundamental. I conclude therefore that any attempt at whole heart pacing must either use fields which are sufficiently uniform to *only* capture fully-recovered tissue—which is not technologically feasible—or use fields which are sufficiently strong to capture all tissue simultaneously—which is what is currently done with countershock.

Until now, it had not been possible to explain these outcomes. Furthermore, from the present results, it seems that multiple-site pacing is unlikely to improve outcome in the foreseeable future.

Chapter 6

Summary and Future Prospects

In this study, I have developed a computer model of the ventricle which is efficient enough to perform a large number of simulations (a statistical set) in reasonable time, while still being reliable enough to provide meaningful conclusions about induction and termination of arrhythmia. I have used this model to explore the local phenomena which can generate arrhythmia and then investigated the induction of arrhythmia on a hypothetical "diseased" myocardial substrate. With induced arrhythmia as a starting point, I have evaluated the utility of proposed anti-arrhythmia pacing techniques.

In chapter 1, I suggested that one measure of the usefulness of a model was the extent to which it decreased reliance on other models. The original purpose of this model was to decrease reliance on animal studies to evaluate anti-arrhythmia pacing. The presumption was that results from the simulations would be used to optimize the protocols before they were tested in animals. This would presumably increase the efficiency of experiments and decrease the loss of the animals.

The results I obtained from the simulations were surprising. Far from discovering that the proposed "uniform-field" pacing was effective in preventing arrhythmia, I discovered that most attempts to pace the entire substrate at low energy, even prior to the onset of polymorphic arrhythmia, were in fact arrhythmogenic. Only those pacing

stimuli that either captured *no* partially-excitable tissue, or virtually *all* partially-excitable tissue prevented more arrhythmias than they caused.

Although the protocols were successful at low and high fields, they relied on the perfect uniformity of the pacing field which would not be a practical requirement for a real device in the foreseeable future. The consequences of the inevitable non-uniformity of real fields are seen in the unacceptable rate of arrhythmogenesis for intermediate-strength fields (Figure 5-6). The most likely reason for this is contained in the material presented in chapter 4: excitation of partially-excited tissue, as will always happen with intermediate-strength pacing fields, tends to *generate* arrhythmia by facilitating fractionation and wavefront-obstacle separation. This tends to generate large numbers of independent wavelets, leading often to sustained polymorphic reentry.

The result I found is consistent with results from a previous experimental multi-site pacing study [17]. In that study, however, there was no explanation of the failure—the animal model provided insufficient information to determine a probable failure mode. Simulations provide complete and continuous information about the state of the model. By inspecting the details of the simulation, a failure mode was easily determined. Since that failure mode is consistent with theoretically predicted and experimentally observed behavior, it seems plausible that a new round of animal experiments, had they been performed, would also have failed.

The simulator clearly decreased reliance on the animal model. More importantly though, it provided information that, in all likelihood, the animal model would not have provided. This demonstrates the importance of another way of viewing the usefulness of a model. Certain models (e.g. animal models) provide a higher degree of "reliability" (fidelity to the process under study), but a lower degree of information (understanding about the underlying processes). Though computer models may never rival the physiologic fidelity of animal models, if they are constructed carefully they

can provide sufficient fidelity to be a source of substantially more understanding because of their greater simplicity and transparency. The value of this model is not only in decreasing reliance on animal models, but also in providing enhanced understanding relative to animal models.

In the remainder of this chapter, I will discuss future studies that could extend the usefulness of this modeling technique. I will discuss two general issues. First, I will discuss experimental studies designed to increased the level of understanding on which the model is based, and thus increase the range of phenomena that can be studied with the model. Then I will discuss extensions of the model itself.

6.1 Proposed Experimental Studies

As described in Section 1.5.3, experimental studies of animals are still very limited in the quantitative information they can provide. This is in part because even a very detailed mapping study such as [30] can only observe the two-dimensional manifestations of three-dimensional phenomena. Experimental studies of the qualitative phenomena predicted in this study should be constructed specifically to explore fundamental aspects of myocardium as an excitable medium. In this section, I will provide some suggestions of the types of studies that could give experimental information directly applicable to this modeling approach.

6.1.1 Hypotheses to be Tested

I propose experiments to test several fundamental assumptions and predictions of my model:

1. **Speed *vs.* Curvature Relationship.** The propagation speed of excitation wavefronts changes linearly with the curvature of the wavefront (Equation 2.23).

2. **Continuous Speed *vs.* Excitability Relationship.** The propagation speed of excitation wavefronts is lower in tissue of lower excitability.

3. **Wavefront-obstacle Separation.** Excitation wavefronts "stick" to obstacles in myocardial tissue with normal excitability. In tissue of lower excitability, the wavefronts separate from the obstacles.

4. **Excitability *vs.* Density of Viabile Tissue.** If the density of viable cells in tissue is decreased, the excitability (and therefore wavefront propagation speed) is also decreased.

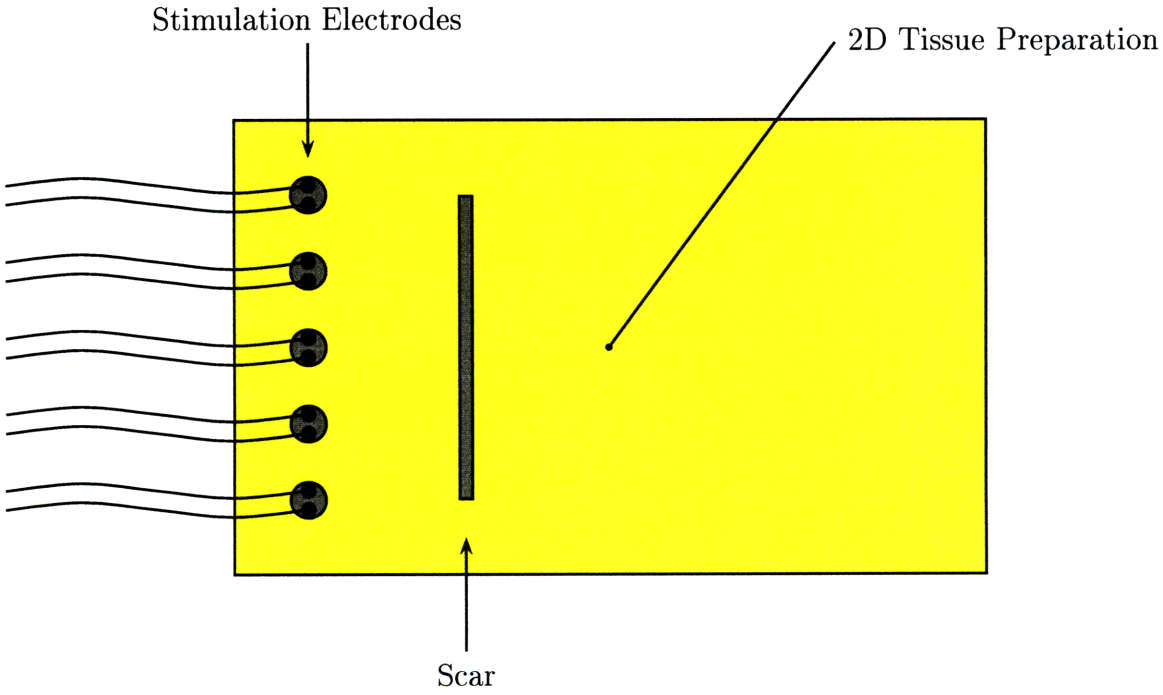
5. **Uniform-field pacing.** The overall prediction of this study is that uniform pacing into a PVC will not decrease the probability of that PVC resulting in sustained arrhythmia. I will propose experiments to test all of these relationships.

6.1.2 Experimental Configurations

Because three-dimensional observations are difficult, I suggest preparations which are effectively two-dimensional. There are at least three experimental substrates that would facilitate this. The first is an *in vitro* atrial tissue preparation (used, for example, in [45]). The second is a thin layer of ventricular tissue prepared by cryogenically destroying most of the thickness of the tissue (for example, [61]). The third is an artificially-grown sheet of cardiac myocytes (for example, [15] or [71]).

The experimental configuration is shown in figure 6-1. Bipolar stimulation electrodes are separated by no more than 1 cm near one edge of the tissue so that the circular waves simultaneously initiated at the individual electrodes will merge into plane waves within a few centimeters. The sample is stained with a voltage-sensitive fluorescent dye, and wavefront location is recorded continuously by a CCD camera [30].

Figure 6-1 Minimal experimental configuration for testing the wave phenomena predicted by this study. All experiments are performed on a two-dimensional sample so the location of the wavefront is always well-defined. An array of pacing electrodes can be used simultaneously to create approximately planar waves. A single scar can be introduced to test separation phenomena. Continuous recording of wavefront position can be done with voltage-sensitive dyes and a CCD camera, using the methods of [30].



A single scar can be created by destroying a thin line of tissue several centimeters long and approximately 2–3 millimeters wide so that the wavefront would be unable to "jump" over the scar. The scar would serve as an obstacle for the study of separation.

6.1.3 Speed-Curvature relationship

The speed-curvature relationship (hypothesis 1) is difficult to measure directly due, for example, to the difficulty of defining a single, unchanging curvature for a propagating wavefront [24]. An approximate speed-curvature relationship can be measured as follows:

1. Begin with the configuration shown in figure 6-1, but with no scar.
2. Pace from a single electrode and record the wavefront position continuously.
3. Plot instantaneous propagation speed at each measurement against inverse distance from the stimulation electrode (curvature).

6.1.4 Continuous Control of Propagation Speed with Tissue Excitability

As the excitability of the tissue is lowered from normal, the propagation speed should decrease continuously until propagation is blocked (hypothesis 2). This can be established with one of two similar experiments:

- **Pacing into Partially-Recovered Tissue.**
  1. Start with the experimental configuration shown in figure 6-1 (no scar).
  2. Create plane waves by pacing simultaneously at all electrodes.
  3. Determine the timing of a train of stimuli, $S_1, \ldots, S_n$, so that each stimulus, S_i is initiated at the earliest time it will capture, and the final stimuli, $S_{n-j}, \ldots, S_n$ are at a constant rate (steady-state)

4. Vary the excitability by gradually increasing the period of the last several stimuli $S_{n-j}, \ldots, S_n$, from the minimum capture period.

5. Continuously record the positions of the wavefronts resulting from the last several stimuli, to determine the plane wave velocities.

- **Pharamacoligic Lowering of Excitability.**

1. Start with the experimental configuration shown in figure 6-1 (no scar).

2. Create plane waves by pacing simultaneously at all electrodes.

3. Lower the excitability of the tissue by administration of a fast Na^+-channel blocking agent, either an antiarrhythmic such as lidocaine which decreases conduction speed, or a very small dose of a Na^+-channel poison such as tetrodatoxin.

4. Record the positions of the wavefront to determine the plane wave velocities as a function of dosage.

6.1.5 Wavefront-obstacle Separation

The phenomenon of wavefront-obstacle separation (hypothesis 3), has been studied experimentally in excitable media [5, 66], but these studies have been conducted in simpler excitable systems such as the Belousov-Zhabotinskii reaction rather than in myocardial tissue. Although the observation of separation phenomena in excitable systems and in PDE models provides important confirmation of the effect, it would be desirable to observe and quantify these effects in myocardial tissue. They can be studied experimentally as follows:

1. Begin with the configuration shown in figure 6-1, including the scar.

2. Lower the excitability of the tissue by either of the methods described above (Sec. 6.1.4).

3. Record continuous wavefront positions at each tissue excitability

4. Determine the excitability (plane-wave propagation speed, from sec. 6.1.4) at which the wavefront first detaches from the scar.

5. Determine whether there is sufficient separation at minimum tissue excitability (the lowest excitability allowing propagation of plane waves) to establish self-sustaining rotors after the separation.

6.1.6 Excitability *vs.* Density of Viabile Tissue

Tissue excitability is decreased if the density of viable cells is reduced, but still essentially uniform on a length scale similar to the transition region wavelength (hypothesis 4). This could be tested on an experimental preparation of artificially-grown myocytes. These preparations have histologic properties that are sensitively dependent on the exact conditions under which they were grown [15], so it may be possible to sample a fairly large range of viable tissue densities by performing experiments on tissue with different preparations [22]. This could be done as follows:

1. Develop several tissue samples under conditions that ensure different histologies.

2. Integrate each sample into the experimental configuration shown in figure 6-1 (no scar).

3. Measure the plane wave speed using the method described above (Sec. 6.1.4).

4. Determine the approximate density of viable tissue from histology.

5. Plot the relationship between measured plane wave speed and tissue density.

6.1.7 Pacing Studies

This study was originally developed to test the anti-arrhythmic effect of uniform-field pacing. Although the modeling results suggest this technique is unlikely to work (hypothesis 5), it may still be interesting to test it experimentally. An experiment could be conducted as follows:

1. Begin with the heart of a large mammal (swine or dog), either *in vivo* (open-chest) or *in vitro* (a perfused preparation), and stain the heart with voltage-sensitive dye.

2. Insert an endocardial coil electrode into the right ventricle.

3. Sew pacing electrodes onto the epicardium of the right atrium and left ventricle.

4. Monitor the activity of the epicardium of the left ventricle with the CCD camera, following [30].

5. Determine the diastolic threshold of the pacing coil (referenced to a metal plate under the animal) by increasing the coil pacing voltage until the region of epicardium furthest from the coil is excited.

6. Following the procedure of section 5.1, determine the refractory period for the ventricular pacing electrode after a rapid train of atrial beats.

7. Apply a "premature" ventricular depolarization, at the end of the refractory period, through the ventricular electrode.

8. Increase the DC current of the ventricular depolarization until the VF induction rate is approximately 50%.

9. Rerun the above protocol, pacing through the coil 80 msec after the ventricular stimulus.

10. Determine the new induction rate.

11. When the technique fails (sustained arrhythmia), determine the starting location of the arrhythmia as the earliest point of epicardial breakthrough.

My modeling study predicts that:

1. The preemptive pacing interventions would *not* decrease the incidence of arrhythmia; and

2. The coil intervention should trigger arrhythmia at intermediate distances from the coil, i.e. where the field is 25–50% above diastolic threshold.

6.2 Future Modeling Directions

In section 1.5, I argued that CA models would be the models of choice far into the future. The model used in this study was successful in modeling arrhythmia and the simplified, hypothetical case of uniform-field pacing. There are many extensions that can be added to the model to increase the range of phenomena that can be investigated by CA techniques. In many respects, the model used here should be considered the simplest realistic implementation of a ventricle and taken as only the beginning of cardiac CA model development. In the next few sections, I suggest some of the improvements that could immediately be made to my implementation of the cardiac CA model.

6.2.1 Three-dimensional Structures

One of the most significant limitations of the current model is the representation of the ventricle as a two-dimensional structures. Although this representation was adequate for the study of the effects of uniform fields, there are several aspects of

electrophysiology which are intrinsically three-dimensional. These include the rotation of fiber anisotropy and change in refractory period in the ventricular wall, the detailed interaction of electric fields with tissue, and the existence of slow, transmural pathways for reentrant wavelets. Also important is that there is no meaningful way to represent the His/Purkinje system in a two-dimensional model, so true nodal beats could not be modeled.

The original decision to construct the model in two dimensions was based on tradeoff between model size and complexity, which favored two-dimensional modeling, and representation of detailed anatomy, which favor three dimensions. Given the computer resources available at the time, the compromises in detailed representation required by a three-dimensional CA model, e.g. use of larger lattice spacing to allow the model to fit into computer memory, would have been a greater limitation than the lack of a third dimension.

Current computer resources have evolved to the point where a full three-dimensional model would be feasible on a workstation, having approximately 512 MB to 1024 MB of RAM. Simulation speed on a computer such as a DEC Alpha of the three-dimensional model would probably be within a factor of 10 of the simulations speeds obtained in this study on the Intel Pentium Pro.

There is nothing about the techniques developed in chapter 2, or the implementation described in chapter 3, that is specific to two-dimensions. Extension of the model to true three-dimensional geometry would require no architectural changes and only minor changes to the code.

6.2.2 Detailed Electrical Anatomy

The significant portion of the effort in creating a three-dimensional CA model of the heart would be the detailed incorporation of the three-dimensional electrical anatomy of the ventricles. The continuous rotation of fibers through the ventricular wall has

already been described in edge-triggered CA models [72], and could be directly implemented in a three-dimensional implementation of this model.

The parameters of the His/Purkinje system could be implemented using the same techniques as other myocardial tissue. The directionality and high conduction speed in the His/Purkinje system would be implemented as a high plane wave speed and very large anisotropy ratio. This would allow the modeling of true nodal beats.

6.2.3 Electric Fields

A CA model does not, by construction, have the capacity to incorporate the direct effects of electric fields. Electric fields interact directly with the membrane potentials of the individual myocytes, a level of detail which is not available in CA models. As with all other continuous processes in excitable media, however, it is instead necessary to establish correspondence between the behavior of the model and the behavior either of the actual system or of continuous models of the system.

To implement the detailed effects of electric fields, it is therefore necessary to establish the nature of those effects in other models, most probably ion channel models or animal models. Given this information, there are two distinct techniques that can be used to incorporate the information into CA models. First, available information can be incorporated empirically, in analogy to the restitution relationship of myocardial tissue (Section 3.1.1). The would involve the modification of the state transition rules described in section 3.2.1 to allow excitation when the field strength at that location exceeded the local field threshold for that element. The threshold would depend on the state of the element according to a relationship similar to that used for partial excitability. This would allow the incorporation of the observed effects of graded response to high field stimulation. Field inhomogeneity would be modeled by determining *a priori* the field strength at each point in the lattice and incorporating that strength into the local state transition rule.

A second technique for incorporation of electric fields into the model would require a more detailed understanding of the *mechanism* by which the field interacts with the myocyte at various stages of recovery. This type of information would specify the probability of excitation of a myocyte given the combined effects of external field and local excitation current from neighboring myocytes, as a function of the local recovery phase of the element. As above, the local external field strength would be calculated *a priori*. Given this information, it would again be possible to create state transition rules which simultaneously incorporated the effects of excitation wavefronts and external fields.

The first approach, that of using empirical information derived from experiments, e.g. measured strength-interval curves could be implemented in a three-dimensional version of the CA model with very little modification to the model structure. The second approach, incorporating the effect of the field in a physically-consistent manner, will be practical when more detailed information is available about the interaction of external fields with myocytes.

6.3 Conclusion

The model developed in not limited to this study. I used the model to test a very simple pacing hypothesis and that test provided useful information. More sophisticated pacing schemes could be implemented. For example, local stimulation synchronized to local electrograms, or pacing interventions with different models of disease. Such studies would not be realistic in a two-dimensional implementation of the model, since transmural propagation away from the stimulation electrodes would be a significant factor, but the study would be easy to conduct in a three-dimensional implementation of the model.

The model is also not limited to the study of pacing. The results presented in

chapter 4 gave a very useful picture of the generation of arrhythmia. This model could be used for a more detailed study of arrhythmogenesis under diverse sets of circumstances. In particular, although there is good evidence of the importance of three-dimensional anisotropy in the transition to ventricular fibrillation, it would be possible to study the problem with detailed anisotropy.

This study has provided a model of cardiac electrophysiology and demonstrated the use of that model for guiding experiments in animal models and in gaining understanding of mechanisms. The model presented in this study promises to continue to add to our understanding of cardiac electrophysiology and so to increase the efficiency of experimental studies; decreasing the reliance on animal models, while improving understanding of electrophysiologic mechanisms.

Appendix A

Code

This is a listing of the final version of the simulator code which produced the preceding results. Much of the code that not directly related to the simulation engine, *e.g.* graphics code, code for reading parameters and general data types such as queues, has not been included, since the function is not unusual.

This code was complied on version 2.7 of the gcc complier from the Free Software Foundation and used the general C++ class library libg++ from the same group. All code was run on the Linux version 2 kernel on i386-architecture microprocessors.

A.1 Element.h

A goal in many object-oriented languages, such as `C++`, is the complete definition of data and behavior in a `class`. In the file *Element.h*, I define `class Element`, which is the fundamental class in the simulator—a single finite-state automaton, representing a single "piece" of myocardial tissue. All the data which describe the complete state of an element (Sec.3.2.1), as well as the fundamental routines for manipulation and analysis of that data are defined in `class Element`.

I include this file first because it is included in nearly all the other files in the program, and also because most of the simulator action is on the data and with the routines defined in this file. Much of the code in the rest of the simulator is "support" for this file.

```
#include <stdlib.h>

inline int round (float x) { return (int)(x+0.5); }

#define SAFE
#define FIX_HOOD

# define SPEED def_speed   // Default conduction speed in cm / sec
# define MAX_PROB 0xFF // Maximum value for interaction probability
# define MAXSPEED 100 // Maximum propagation speed attainable by simulator in
                      //   cm/sec
# define MAXRAD 1

/*
  RELREF, EXCITABLE, ACTIVE, and REFRACTORY are states characterizing
  viable elements.  SCAR is a "dead" (permanently unexcitable) element, and
  DUMMY is an element that should not be considered part of the lattice
  (those "cut out" of a lattice which is intrinsically rectangular–see the
  comment at the top of the file Lattice.h (Sec. A.6)
*/
enum Elem_states { RELREF = -1, EXCITABLE, ACTIVE, REFRACTORY, SCAR, DUMMY,
                   LAST_STATE };
/*
  The neighbors structure contains an instance of a neighborhood, which is all
  the locations of the elements that can be effected by this element.  In
  practice, the neighborhood implicitly specifies the speed anisotropy ratio
  (Section 2.6).  The statistical properties of the
  possible neighborhood instances also provide
  statistically-symmetric wave propagation.  The details are in
  sec. A.4 and sec. A.3)
*/
#ifdef FIX_HOOD
struct neighbors;
#endif
class Hood_dist;
struct State_info;

extern int thresh [(MAXSPEED + 1)]; // table relating threshold to speed
extern float rads [(MAXSPEED + 1)]; // table relating radius to speed
extern unsigned int excit_power[256]; // decay of power with time
extern int def_speed, min_speed, max_speed;
extern int active_time;              // duration of EXCITING state

/*
  Steps in evaluating the next state of viable elements:
    if (state) eval_next_state() :
        if (REFRACTORY) refractory_update();
        else if (RELREF) relref_update();
        else active_update();
    if (RELREF) check_recovery_level();
    update();
```

```
*/

class Element {
    char curstate, nextstate;
    unsigned short di;          // diastolic interval (ticks)
    int excit;                  // Excitation power received (this tick)
    int loc_thresh;             // ecitation threshold
    Element *left, *right, *front, *back;      // links to nearest neighbors
#ifdef FIX_HOOD                                                          60
    neighbors *hood;            // Neighborhood for this excitation cycle
#endif
    Hood_dist *hoods;           // Distribution from which to choose hood
    short timer;                // how long remaining in current state
    short tau;                  // time constant for history dependence of
                                //   action potential duration (const)
    short apd, last_apd;        // base action potential duration (const)
    unsigned char base_speed; // base propagation speed
    // State-dependent functions called to update the state of the element
    // Defined in (Sec. A.2)                                             70
    inline void active_update (void);
    inline void refractory_update (void);
    inline void relref_update (void);
  public:
    // Function called to update state of Element (Sec. A.2)
    inline void eval_next_state (void);
    // Function called to "fix" Element state after updating
    int update() { excit = 0; return curstate = nextstate; }
    // activate() and the check_neighborhood() routines are defined in
    // (Sec. A.4)                                                        80
    int activate (void);        // set element to EXCITING state (if possible)
    // check_neighborhood() is called by an EXCITING element to deliver
    // excitation power to every element in its neighborhood
    void check_neighborhood (int power, int flag);
    void check_neighborhood (int power);
    // stimulate() is called from check_neighborhood on a particular element in
    // an element's neighborhood that is currently excitable.  It is called  to
    // provide stimulation (power) to this element
    void stimulate (int power) {
        if ((excit += power) >= loc_thresh) activate();                  90
    }
    // set_lsp() is called to change the local propagation speed as an element
    // progresses through the RELREF state (Sec. ~\ref{sec:mainsim.cc})
    inline void set_lsp (int speed);
    // set the local propagation speed to baselien speed
    inline void reset_lsp (void);
    void set_lsp (void);
    // The constructor, which initializes the element at lattice creation
    Element (void) {
        di = 10000; curstate = EXCITABLE;                                100
#ifdef FIX_HOOD
        hood = 0;
```

```
#endif
    }
    // Find the nearest-neighbor elements
    Element *find_left (void) { return left; }
    Element *find_right (void) { return right; }
    Element *find_front (void) { return front; }
    Element *find_back (void) { return back; }
    // This routine designates an element as not participating in activity
    void init_dummy (void) { loc_thresh = -1; curstate = nextstate = DUMMY; }
    void set_base_ref(int val) { apd = last_apd = val; }
    int get_base_ref() { return apd; }
    int get_apd (void) { return last_apd; }
    void set_base_speed (int val) { base_speed = val; set_lsp(); }
    void set_timer(int val) { timer = val; }
    int get_timer() { return timer; }
    void set_hood_dist (Hood_dist *hd) { hoods = hd; }
    void reset_excit() { excit = 0; }
    int get_tau (void) { return tau; }
    void set_tau(int val) { tau = val; }
    int state_is() { return curstate; }
    int is_viable() { return curstate <= REFRACTORY; }
    int IS_RESTING() { return ! curstate; }
    int IS_RELREF() { return curstate < 0; }
    int IS_ACTIVE() { return curstate == ACTIVE; }
    int is_dummy() { return curstate == DUMMY; }
    // is_depolarized gives a cutoff beyond which an element is considered
    // 'depolarized' for EKG purposes
    int is_depolarized (void) { return curstate > 0 && is_viable(); }
    int is_recovering (void) { return curstate != EXCITABLE && is_viable(); }
    int is_excitable() {
        return (curstate <= 0 && /* nextstate <= 0 && */ loc_thresh > 0);
    }
    int will_excite() { return nextstate == ACTIVE; }
    int get_stim() { return excit; }
    int get_thresh() { return loc_thresh; }
    int watch_ref (void) {
        if (nextstate == REFRACTORY && curstate != REFRACTORY)
            return timer;
        return 0;
    }
    // This is a spy routine defined in lattice.cc for debugging purposes
    int watch_elem (void);
    // return the location of the fiducial point of an element
    int get_center_offset (float &x, float &y);
    void make_scar (void) {
        curstate = nextstate = SCAR;
    }
    void make_refractory_edge (void) { nextstate = REFRACTORY; timer = apd; }
    void make_refractory_edge (int ticks) {
        nextstate = REFRACTORY; timer = ticks;
    }
```

```
    // traverse_neighborhood() is like check_neighbors(), but calls the
    // function specified in arg 1 on each element in the neighborhood
    int traverse_neighborhood (int (*) (Element *), Hood_dist *n = 0);
    int pace (void);

friend class Lattice;
friend void put_ds (class Graphic *);                                    160
friend void update_ds (class Graphic *);
friend void update_ds (class Graphic *, int);
friend void do_print (int cnt);
friend float *disease (float sdev, int, int, float, float, long& idum);
friend int beat (int);
friend void watch_element (void);
};
```

A.2 mainsim.cc

Most of the routines defined in `class Element` are defined in the file *mainsim.cc*, as is the `main()` function which starts the simulator. The main iteration routine is `beat()`, which is called in a loop from `main()`. This file also contains the routines responsible for forking subprocesses.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include "Queue.h"
#include "Element.h"
#include "Params.h"
#include "logfile.h"
#include "APD.h"
#include "Lattice.h"
#include "interface.h"
#include "Neighbors.h"
#include "GeomObj.h"
#include "Pace.h"
#include "Process.h"

int cnt;
int child = 0; // true if child process

// All the externally applied "beats" (pacing events) are stored in
// beat_queue.
static Beat_queue beat_queue;

/* $Format: "char *prog_name = \"$ProjectHeader$\";"$ */
char *prog_name = "lsim 4.2 Fri, 06 Feb 1998 17:39:07 -0500 paul";

// Graphics display routines
void put_ds (void), update_ds (int);
void plot_excit_power (int);

inline void Element::set_lsp(int speed)
{
    loc_thresh = hoods->get_threshold (speed);
}

inline void Element::reset_lsp (void)
{
    set_lsp (base_speed);
```

```
}

void Element::set_lsp (void)
{
    reset_lsp();
}

inline void Element::active_update (void)
{
    --timer;
    check_neighborhood (excit_power[timer]);
    if (timer <= 0) {
        nextstate = REFRACTORY;
        timer = last_apd = (*Restitution) (di, tau, apd);
    }
}

inline void Element::refractory_update (void)
{
    if (--timer <= 0) {
        nextstate = RELREF; timer = relref_time;  di = 0;
        set_lsp (0);
    }
}

inline void Element::relref_update (void)
{
    if (--timer <= 0 && nextstate == RELREF) {
        nextstate = EXCITABLE; ++di; set_lsp (base_speed);
    }
    else { set_lsp ((*Dispersion) (timer)); ++di; }
}

// this function changes the timers of refractory and active elements and
// controls the interactions between active elements and their neighborhoods.
void Element::eval_next_state (void)
{
    if (is_dummy()) return;
    switch (curstate) {
        case EXCITABLE :   ++di; return;
        case REFRACTORY :  refractory_update(); return;
        case RELREF :      relref_update(); return;
        case ACTIVE :      active_update(); return;
    }
}

enum CONTROL_STATES next_state (enum CONTROL_STATES, int&, int&);
int do_beats (int), do_trips (int);
int do_ekg_p (int);
void start_ekg (int), ekg_line (int, int), put_ekg (int);
void put_movie (int);
```

```
void put_time (int iter, Graphic *g = 0);
void plot_dispersion (int, float);
float update_history (int, Beat_queue&);
void put_loc (int x, int y, int size = 5);
void do_circle (float phi_c, float theta_c, float r_c, void (*) (Element *));

// beat() is the main iteration routine in the simulator.  It is called in
// an "infinite" loop from main().  The button argument is no longer
// used.
int beat (int button)                                                    100
// This function is called once per iteration and controls the program
// after initialization.
{
    register Element *te, *le ;
    // The "state" of the simulator is controlled interactively by the user
    static enum CONTROL_STATES state;
    int laststate = state, recovering, active;
    int i, j, maxsize, x, y;
    int edge_x = lattice->get_x_len(), edge_y = lattice->get_y_len();
    char eflag;                                                          110
    int depol_num;
    int beats, trips;
    extern float time_step;
    Pace *check_capture (int cnt, int active), *last_beat;
    extern float apd_max;
    static int *depol_cols = 0;
    int *c;
    extern int user_abort;
    int init_circle(float x, float y, float size);
    void draw_elem (Element *e);                                         120
    float xloc, yloc;

    state = next_state (state, x, y);
    if (state == EXIT_SIM) {
        log_val ("User-terminate", cnt);
        printf (" terminated.");
        return 0;
    }
    else if (state == INTERRUPT_SIM) {
        log_val ("Signal-terminate", cnt);                               130
        printf (" terminated.");
        return 0;
    }
    if ((state & TOGGLE_SIM) && one_shot (state) != SINGLE_STEP) {
        usleep (100000); return 1;
    }
    if (one_shot (state) == GOT_POINT) {
        te = (*lattice) (x, y);
        lattice->location (te, xloc, yloc);
        init_circle (xloc, yloc, 0.4);                                   140
    }
```

```
    ++cnt;    // iteration count.  This is the only timer in the simulator

    beats = do_beats (cnt);   // execute any scheduled beats
    trips = do_trips (cnt);   // execute any scheduled measurement events

    for (te = (*lattice)(), le = lattice->lattice_end();  te < le;  ++te)
        te->eval_next_state (); // preliminary latticde update

    if (! (state & TOGGLE_DISPLAY)) update_ds (cnt); // graphics update
    put_time (cnt);           // time update

    // collect lattice statistics on this iteration
    for (te = (*lattice)(), y = 0, active = recovering = 0;
         te < lattice->lattice_end();  ++y)
        for (x = lattice->get_x_len(), c = depol_cols;  --x >= 0;  ++te, ++c) {
            switch (te->update()) {
                case ACTIVE : ++active;
                case RELREF :
                case REFRACTORY :
                        ++recovering; break;
            }
            //      if (te->is_depolarized()) ++*c;
        }

    // execute any scheduled events to determine if beats were captured
    if (last_beat = check_capture (cnt, active)) {
        beat_queue += new Beat_time (last_beat->get_current());
        //      update_history (cnt, beat_queue);
    }

    put_movie (cnt);  // conditionally output movie frame of iteration

    // simulation expires when no queued beats and not activity
    if (beats < 0 && ! active) {
        printf ("Simulation expired at %4d (%6.3f s).", cnt, cnt * time_step);
        log_val ("Expired", cnt);
        return 0;
    }
    if (! (cnt & 0x1F)) { putchar ('.'); fflush (stdout); }
    return 1;
}

char *output_name;

void poke_element (int i, int j)
{
    Element *te;
    int edge_x = lattice->get_x_len(), edge_y = lattice->get_y_len();

    for ( ;  (te = (*lattice) (i, j))->is_dummy();  ++i)  ;
    getchar();
```

```
    te->activate();
    put_ds();  te->update(); getchar(); put_ds();
    printf ("(%d, %d) before check. . .", i, j); fflush (stdout); getchar();
    te->check_neighborhood (1);
    for (te = (*lattice)(), j = 0; j < edge_y;  ++j)
        for (i = 0; i < edge_x; ++i, ++te) te->update();

    put_ds(); printf ("after check.\n"); fflush (stdout); getchar();
}

/*
  Subprocesses are forked to allow self-controlled simulation experiments.
  The parent process waits for the child to complete.  The child uses a pipe to
  pass back a summary of its run before it exits in a Sim_info structure.
  This allows the parent to determine what was captured in the child process
  and to determine is there was self-sustained activity.
*/
int pipe_des[2] = {-1, -1};
int get_frame_num (void);

// returns 0 for child, else 1
Sim_info *fork_sim (void)
{
    pid_t pid;
    int status;
    Sim_info *child_info;

    if (pipe_des[0] >= 0) close (pipe_des[0]);
    if (pipe (pipe_des) < 0) {
        fprintf (stderr, "pipe failed.\n");  return new Sim_info;
    }
    sim_info_struct->start_cnt = cnt;
    sim_info_struct->start_frame = get_frame_num();
    log_val ("fork_frame", sim_info_struct->start_frame);
    flush_log();
    pid = fork();
    if (pid < 0) {
        fprintf (stderr, "fork failed.\n");  return new Sim_info;
    }
    else if (pid) {             // parent
        close (pipe_des[1]);    // close write descriptor
        wait (&status);
        flush_log();
        child_info = new Sim_info;
        read (pipe_des[0], child_info, sizeof(Sim_info));
        log_val ("child_cnt", child_info->end_cnt);
        log_val ("child_frame", child_info->end_frame);
        log_val ("return", cnt);
        put_ds();
        return child_info;
    }
```

```
    else {
        close (pipe_des[0]);     // close read descriptor
        log_val ("fork", cnt); printf ("fork "); fflush (stdout);
        child = 1; return 0;
    }
}
                                                                    250
Sim_info *sim_info_struct;

main(int argc, char *argv[])
{
    int i, j, sc, argn;
    void mainloop (int (*fn)(int));
    FILE *pfp;
    int s;
    int scnt;
    char *input_ckp = NULL, *log_name = NULL;                       260
    void make_lattice (void), setup_apd (void), set_times (void),
         set_propagate (void), setup_substrate (void), setup_events (void),
         setup_screen (void), setup_files (void), put_params (void);
    void got_user_abort (int);
    int restore_ckp (char *);
    int out_params = 0;
    int init_only = 0;
    Element *te;

    sim_info_struct = new Sim_info;                                 270
    sim_info_struct->start_cnt = sim_info_struct->start_frame = 0;

    if (argc <= 2) {
        fprintf(stderr,
                "usage: lsim output [-l log] [-c checkpoint] params (-)\n");
        exit(1);
    }
    printf ("%s\n", prog_name);
    if (*argv[1] == '-') output_name = NULL; else output_name = argv[1];
    for (argn = 2;  argn < argc;  ++argn)                           280
        // restore state from checkpoint file
        if (strcmp (argv[argn], "-c") == 0) {
            ++argn;
            if (argc <= argn) {
                fprintf (stderr, "failure to specify checkpoint input file\n");
                exit (1);
            }
            input_ckp = argv[argn];
            continue;
        }                                                           290
        else if (strcmp (argv[argn], "-l") == 0) {
            ++argn;
            if (argc <= argn) {
                fprintf (stderr, "failure to specify checkpoint input file\n");
```

```
            exit (1);
        }
        log_name = argv[argn];
        continue;
    }
    else if (strcmp (argv[argn], "-p") == 0) {
        out_params = 1;
    }
    else if (strcmp (argv[argn], "-I") == 0) {
        init_only = 1;
    }
    else if (strcmp (argv[argn], "-") == 0) new_param_file (stdin);
    else {
        if (! (pfp = fopen (argv[argn], "r"))) {
            fprintf (stderr, "Unable to open param file %s for read.\n",
                    argv[argn]);
            exit (1);
        }
        new_param_file (pfp);
    }

if (log_name)
    if (! open_log (log_name)) {
        fprintf (stderr, "Logging error.\n");
        exit (1);
    }

signal (SIGINT, got_user_abort);
start_log (prog_name);

new_initfn (setup_apd);
new_initfn (make_lattice);
new_initfn (setup_screen);
new_initfn (set_propagate);
new_initfn (set_times);
new_initfn (setup_substrate);
new_initfn (setup_events);
new_initfn (setup_files);

get_params();

if (out_params) { put_params(); exit (0); }

if (cnt) put_ds();

if (init_only) { putchar ('\n'); return 0; }

while (beat (1)) ;

if (child) {
    sim_info_struct->end_cnt = cnt;
```

```
        sim_info_struct->end_frame = get_frame_num();
        write (pipe_des[1], sim_info_struct, sizeof(Sim_info));
    }
    end_log();
    putchar ('\n');
}
```

A.3 Neighbors.h

An element interacts with all other elements in its "neighborhood." An element's neighborhood is defined in `struct neighbors`. Each instance of `struct neighbors` contains a single neighborhood. A statistical distribution of those neighborhoods allows symmetric propagation of waves (Sec.2.3.2). These structures are defined in the file *Neighbors.h*. The code that allows an element to access each element in its neighborhood is in the file *neighbors.cc*.

```
#ifndef NEIGHBORS_H
#define NEIGHBORS_H

#include "Distribution.h"

/*
  n_line is the fundamental unit of a neighborhood.  It represents those
  elements in a particular row of the lattice which are included in a
  particular instance of a neighborhood ensemble.
  */
struct n_line {
    static unsigned char *n_space, *last_n_space;
    static const int N_SPACE_ALLOC;
    struct n_line *next;        // n_lines are a singly-linked list
    char skip, length;          // start at skip from current and go length
    unsigned char *line;        // bitmask of elements in neighborhood
    n_line (int len, char *p);
    static unsigned char *get_line (size_t len);
    void *operator new (size_t size);
};

// Allocation of n_lines is done by brute force to minimize overhead, and
// since they are never freed.

// get_line() is the basic allocation routine used for structure and bitmask
inline unsigned char *n_line::get_line (size_t len)
{
    unsigned char *line;

    if (n_space + len >= last_n_space) {
        if (! (n_space = ::new unsigned char [N_SPACE_ALLOC]))
            printf ("n_line allocate failure!\n");
        last_n_space = n_space + N_SPACE_ALLOC;
    }
    line = n_space;
```

```
    n_space += len;
    return line;
}

inline void *n_line::operator new (size_t size)
{
    return (void *)n_line::get_line (size);
}

// The neighbors struct holds all the n_lines that represent a particular
// instance of a neighborhood. The lines above (and including) the element are
// stored as a singly-linked list at up. The lines below the element are a
// singly-linked list at down.
struct neighbors {
    struct n_line *up, *down;
    float xoff, yoff;    // the fiducial point of the central element
    neighbors (void) { up = down = 0; }
    neighbors (float x, float y) : xoff (x), yoff (y) { up = down = 0; }
};

// The Hood_dist class contains a complete statistical set of neighborhoods.
class Hood_dist {
    int num_hoods;          // number of neighborhood members in ensemble
    neighbors **hoods;      // array of neighborhood members
    RNG *rgen;              // random number generator to select members
    int bits;               // quick mask to randomly select members
    float vrad, hrad;       // radius at which neighhborhoods generated
    short dir1, dir2;       // directions of neighborhoods
    int *thresholds;        // thresholds to map speeds to neighborhoods
public:
    Hood_dist::Hood_dist (int nhoods, float haxis, float vaxis,
                          float scatter = 1.0, int rot1 = 0, int rot2 = 0);
    // the function operator returns a randomly-selected neighborhood
    neighbors *operator() (void) {
        int n = rgen->asLong();
        if (bits >= 0) return hoods[n & bits];
        else return hoods[n % num_hoods];
    }
    // when called with an integer argument, n, the function operator returns
    // the nth neighborhood in the set
    neighbors *operator() (int n) {
        return hoods[n % num_hoods];
    }
    int get_threshold (int speed) { return thresholds[speed]; }
    int compare (Hood_dist& h2) {
        if (vrad < h2.vrad) return -1;
        else if (vrad > h2.vrad) return 1;
        else if (hrad < h2.hrad) return -1;
        else if (hrad > h2.hrad) return 1;
        else if (dir1 < h2.dir1) return -1;
        else if (dir1 > h2.dir1) return -1;
```

```
        else if (dir2 < h2.dir2) return -1;
        else if (dir2 > h2.dir2) return 1;
        else if (num_hoods < h2.num_hoods) return -1;
        else return 0;                                                    90
    }
    int compare (float haxis, float vaxis, int n = 0,
                 int rot1 = 0, int rot2 = 0) {
        if (vrad < vaxis) return -1;
        else if (vrad > vaxis) return 1;
        else if (hrad < haxis) return -1;
        else if (hrad > haxis) return 1;
        else if (dir1 < rot1) return -1;
        else if (dir1 > rot1) return 1;
        else if (dir2 < rot2) return -1;                                  100
        else if (dir2 > rot2) return 1;
        else if (num_hoods < n) return -1;
        return 0;
    }
friend Hood_dist *get_hoods (int, float, float, float = 1.0, int = 0, int = 0);
    friend class Element;
};

#endif
```

A.4 neighbors.cc

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "Element.h"
#include "Lattice.h"
#include "APD.h"
#include "SortQueue.h"
#include "Neighbors.h"
#include "Distribution.h"
#ifdef TEST                                                          10
#include <unistd.h>
#include "Graphic.h"
#endif

unsigned char *n_line::n_space = 0;
unsigned char *n_line::last_n_space = 0;
const int n_line::N_SPACE_ALLOC = 1 << 14;

// The constructor for n_line takes the calculated neighborhood line in
// "uncompressed form," that is, an array at least long enough to hold the   20
// line, with one element per byte.  It returns a compressed array with one
// element per bit, and an offset and length showing where the line starts
// releative to a point directly above (below) the central element.
n_line::n_line (int len, char *p)
{
    char *arr = p + len / 2;
    unsigned char *l;
    int end, i, mask;

    for (skip = -len / 2;  skip < len / 2;  ++skip) if (arr[skip]) break;   30
    for (end = len / 2;  --end >= skip;  ) if (arr[end]) break;

    if (skip > len / 2) {
        printf ("%s:%d.  Empty line\n", __FILE__, __LINE__);
        return;
    }
    len = (end - skip + 1) / 8 + 1;
    line = l = new unsigned char [len];  mask = 1;

    for (i = skip, *l = 0;  i <= end;  ++i) {                            40
        if (arr[i]) *l |= mask;
        if (mask == 0x80) { ++l, *l = 0; mask = 1; }
        else mask <<= 1;
    }
    length = end - skip + 1;
    next = 0;
    if (! line) printf ("no line at alloc.\n");
```

```
}

void print_line (n_line& n)
{
    int i, mask;
    unsigned char *l;

    for (i = 0;  i < 40 + n.skip;  ++i) putchar (' ');
    for (mask = 1, l = n.line, i = 0;  i < n.length;  ) {
        if (*l & mask) putchar ('*'); else putchar (' ');
        if (mask == 0x80) { mask = 1; ++l; }
        else mask <<= 1;
    }
    putchar ('\n');
}

SortQueue<Hood_dist> all_hoods;

int Element::get_center_offset (float &x, float &y)
{
    if (! hood) return 0;
    x = hood->xoff; y = hood->yoff;
}

void print_up (n_line *n)
{
    if (n->next) print_up (n->next);
    print_line (*n);
}

void print_hood (neighbors& ngh)
{
    n_line *n;

    print_up (ngh.up);
    for (n = ngh.down;  n;  n = n->next) print_line (*n);
}

static void rotate (float &a, float &c, float &b, float dir)
{
    float newa, newc;
    float cs = cos (dir), sn = sin (dir);

    newa = cs * cs * a + sn * sn * c;
    newc = sn * sn * a + cs * cs * c;
    b = 2 * cs * sn * (a - c);
    a = newa; c = newc;
}

// Ellipse is ax^2 + 2bxy + cy^2
// function returns true if (x,y) is inside the specified ellipse
```

```
static int in_ellipse (float a, float c, float b, float x, float y)
{                                                                          100
    if (b != 0) return a * x * x + 2 * b * x * y + c * y * y < 1.0;
    else return a * x * x + c * y * y < 1.0;
}

#ifdef TEST
static Graphic *sc;
void do_ellipse (Graphic *, neighbors *);

#endif
                                                                           110

// Constructor for Hood_dist.  Takes size of ensemble, nhoods, and
// parameters that specify the size and orientation of the elliptical
// neighborhood
Hood_dist::Hood_dist (int nhoods, float haxis, float vaxis,
                      float dist = 1.0, int rot1 = 0, int rot2 = 0) :
    num_hoods (nhoods), rgen (rand_gen),
    vrad (vaxis), hrad (haxis), dir1 ((short)rot1), dir2 ((short)rot2)
{
    int i;                                                                 120
    Uniform random (-dist/2, dist/2, rand_gen);
    neighbors *ngh;
    float a, b, c;
    float xoff, yoff, txoff, tyoff;
    float x, y;
    float nrm_area;
    int xi, yi;
    int grid_len;
    char *grid_buf, *grid_arr;
    n_line *n, *null_n;                                                    130

    hoods = new neighbors * [nhoods];

    if (! (nhoods & (nhoods - 1))) bits = nhoods - 1;
    else bits = -1;

    a = 1 / (haxis * haxis);  c = 1 / (vaxis * vaxis);
    if (rot1 != 0) rotate (a, c, b, rot1 * PI / 180.0); else b = 0;
    grid_len = 2 * (2 + (int)sqrt (haxis * haxis + vaxis * vaxis)) + 1;
    grid_buf = new char [grid_len];                                        140
    grid_arr = grid_buf + grid_len / 2;
    for (i = 0;  i < nhoods;  ++i) {
        xoff = random();      yoff = random();
        hoods[i] = ngh = new neighbors (xoff, yoff);
        for (yi = 0;  yi < grid_len / 2;  ++yi) {
            for (xi = -grid_len / 2;  xi < grid_len / 2;  ++xi) {
                if (xi == 0 && yi == 0) { grid_arr[xi] = 0; continue; }
                txoff = random();      tyoff = random();
                x = xi - xoff + txoff; y = yi - yoff + tyoff;
```

```
            grid_arr[xi] = in_ellipse (a, c, b, x, y);
        }
        if (! ngh->up) n = ngh->up = new n_line (grid_len, grid_buf);
        else {
            for (n = ngh->up;  n->next;  n = n->next)  ;
            n = n->next = new n_line (grid_len, grid_buf);
        }
        if (n->length) null_n = 0;
        else
            if (! null_n) null_n = n;
    }
    if (null_n) {
        for (n = ngh->up;  n && n->next != null_n;  n = n->next)  ;
        if (n) n->next = 0;
    }
    for (yi = -1;  yi > -grid_len / 2;  --yi) {
        for (xi = -grid_len / 2;  xi < grid_len / 2;  ++xi) {
            txoff = random();     tyoff = random();
            x = xi - xoff + txoff; y = yi - yoff + tyoff;
            grid_arr[xi] = in_ellipse (a, c, b, x, y);
        }
        if (! ngh->down) n = ngh->down = new n_line (grid_len, grid_buf);
        else {
            for (n = ngh->down;  n->next;  n = n->next)  ;
            n = n->next = new n_line (grid_len, grid_buf);
        }
        if (n->length) null_n = 0;
        else
            if (! null_n) null_n = n;
    }
    if (null_n) {
        for (n = ngh->down;  n && n->next != null_n;  n = n->next)  ;
        if (n) n->next = 0;
    }
}
thresholds = new int [MAXSPEED+1];
thresholds[0] = 0;
// Re-normalize the thresholds to account for the reduction in area.

// The method of renormilization is a little strange so that it allows for
// setting directional speeds less than or greater than the nominal speed
// for the tissue.  Therefore, if the axis in a given direction is less
// than the nominal axis, the threshold is reduced by that ratio, but if
// the axis is greater than the nominal axis (greater speed in that
// direction), the threshold is not increased.
nrm_area = 1;
if (haxis < rads[def_speed]) nrm_area *= haxis / rads[def_speed];
if (vaxis < rads[def_speed]) nrm_area *= vaxis / rads[def_speed];
for (i = 1;  i <= MAXSPEED;  ++i) {
    thresholds[i] = round (thresh[i] * nrm_area);
}
```

```
    all_hoods += this;
}

// get_hoods() is designed to find a Hood_dist with the specified
// properties if one has already been created, otherwise to create one.
Hood_dist *get_hoods (int nhoods, float haxis, float vaxis,
                      float dist = 1.0, int rot1 = 0, int rot2 = 0)
{
    Hood_dist *h;
    fflush (stdout);                                                    210
    for (h = all_hoods.first();  h;  h = all_hoods.next())
        if (h->compare (haxis, vaxis, nhoods, rot1, rot2) == 0) {
            return h;
        }
    return new Hood_dist (nhoods, haxis, vaxis, dist, rot1, rot2);
}

// Activate an excitable element
int Element::activate (void) {
    int speed;                                                          220
#ifndef TEST
    if (loc_thresh <= 0) return 0;
    if (IS_RESTING()) speed = base_speed;
    else speed = (*Dispersion) (timer);
#endif
    nextstate = ACTIVE;
#ifndef TEST
#ifdef FIX_HOOD
    hood = (*hoods)();
#endif                                                                  230
#endif
    timer = active_time;
    return 1;
}

// Check neighborhood as declared in class Element
void Element::check_neighborhood (int power)
{
    // Rows of interactions are stored as n_lines in the nhood structure, with
    // the rows above (include the row containing this) stored as a linked-list   240
    // from 'up', and the rows below from 'down'.  's' is always the pointer to
    // the "zero-element" of the current row, that is, the element closest to
    // being directly above (below) this, that is used as the index of the
    // row.
    Element *s, *p;
    int left_dev = 0;
    unsigned char *l, mask;
    struct n_line *n;
    int i;
    neighbors *nhood;                                                   250
```

```
#ifdef FIX_HOOD
    if (! hood) hood = (*hoods)();
    nhood = hood;
#else
    nhood = (*hoods)();
#endif

    for (p = s = this, n = nhood->up;  n;  n = n->next) {
        if (n->skip < 0) for (i = -n->skip;  --i >= 0;  )  p = p->left;
        else for (i = n->skip;  --i >= 0;  )  p = p->right;

        for (i = n->length, l = n->line, mask = 1;  --i >= 0;  p = p->right) {
            if ((*l & mask) && p->is_excitable()) {
                p->stimulate (power);
            }
            if (mask == 0x80) { ++l; mask = 1; }
            else mask <<= 1;
        }

        if (! s->back)
            if (left_dev <= 0) while (! s->back) { s = s->left; ++left_dev; }
            else while (! s->back) { s = s->right;  --left_dev; }
        else if (s->back->is_dummy()) break;
        s = s->back;  p = s;
    }
    left_dev = 0;
    for (p = s = this, n = nhood->down;  n;  n = n->next) {
        // We've already done the row containing elem, so immediately move one
        // down.
        if (! s->front)
            if (left_dev <= 0) while (! s->front) { s = s->left; ++left_dev; }
            else while (! s->front) { s = s->right; --left_dev; }
        else if (s->front->is_dummy()) break;
        s = s->front;  p = s;

        if (n->skip < 0) for (i = -n->skip;  --i >= 0;  )  p = p->left;
        else for (i = n->skip;  --i >= 0;  )  p = p->right;

        for (i = n->length, l = n->line, mask = 1;  --i >= 0;  p = p->right) {
            if ((*l & mask) && p->is_excitable()) {
                p->stimulate (power);
            }
            if (mask == 0x80) { ++l; mask = 1; }
            else mask <<= 1;
        }
    }
}

int Element::traverse_neighborhood (int (*fn)(Element *), Hood_dist *hoods = 0)
{
    Element *s, *p;
```

```
    int left_dev = 0;
    unsigned char *l, mask;
    struct n_line *n;
    int i;
    int cnt = 0;
    neighbors *hood;

    if (! hoods) hoods = this->hoods;                                    310
    hood = (*hoods) ();
    for (p = s = this, n = hood->up; n; n = n->next) {
        if (n->skip < 0) for (i = -n->skip; --i >= 0; ) p = p->left;
        else for (i = n->skip; --i >= 0; ) p = p->right;
        for (i = n->length, l = n->line, mask = 1; --i >= 0; p = p->right) {
            if (*l & mask) cnt += (*fn) (p);
            if (mask == 0x80) { ++l; mask = 1; }
            else mask <<= 1;
        }
        if (! s->back)                                                   320
            if (left_dev <= 0) while (! s->back) { s = s->left; ++left_dev; }
            else while (! s->back) { s = s->right; --left_dev; }
        else if (s->back->is_dummy()) break;
        s = s->back; p = s;
    }
    left_dev = 0;
    for (p = s = this, n = hood->down; n; n = n->next) {
        // We've already done the row containing elem, so immediately move one
        // down.
        if (! s->front)                                                  330
            if (left_dev <= 0) while (! s->front) { s = s->left; ++left_dev; }
            else while (! s->front) { s = s->right; --left_dev; }
        else if (s->front->is_dummy()) break;
        s = s->front; p = s;

        if (n->skip < 0) for (i = -n->skip; --i >= 0; ) p = p->left;
        else for (i = n->skip; --i >= 0; ) p = p->right;

        for (i = n->length, l = n->line, mask = 1; --i >= 0; p = p->right) {
            if (*l & mask) cnt += (*fn) (p);                             340
            if (mask == 0x80) { ++l; mask = 1; }
            else mask <<= 1;
        }
    }
    cnt += (*fn) (this);
    return cnt;
}

/*
  setup_neighborhoods() is the function called from params to establish the  350
  default neighborhood for the lattice. This works ONLY because the assumption
  is that the radius does not vary, although it could also be taken as the base
  neighborhood, with all others added as special cases.
```

```
  The parameters are (as specified in the command string l) are:
      neighborhoods <horiz> <vert> <rotation> <scatter>
  where <horiz> and <vert> ϵ[0,1] as multiples of the calculated
  radius, (both 1 if unspecified) and <rotation> ϵ[0,180] (0 if
  unspecified) is in counterclockwise degrees. <scatter> is the width of the
  distribution that gives the "fiducial" point in a lattice element and is
  1.0 by default.
  */
void setup_neighborhoods (char *l)
{
    float horiz = 1.0, vert = 1.0, rotation = 0.0, scatter = 1.6;
    float r;
    const int NHOODS = 1024; // arbitrary ensemble size
    Hood_dist *hoods;
    Element *e;

    sscanf (l, "%f %f %f %f", &horiz, &vert, &rotation, &scatter);
    r = rads[def_speed];
    hoods = new Hood_dist (NHOODS, horiz * r, vert * r,
                           scatter, round (rotation * PI / 180.0));
    for (e = lattice->lattice_start(); e < lattice->lattice_end(); ++e)
        if (! e->is_dummy()) e->set_hood_dist (hoods);

}


#ifdef TEST
// This is code to allow testing of the neighborhood functions.  It is not
// generally used

int active_time = 1;

inline void do_card (Graphic *s, float x, float y)
{
    s->Set_Color (BLUE);
    s->fill_box (x, y, x+1.0, y+1.0);
}

void do_ellipse (Graphic *sc, neighbors *ngh, float r_h, float r_v)
{
    n_line *n;
    float x, y;
    int i, t = 0, mask;
    unsigned char *l;

    sc->Set_Color (WHITE);
    sc->fill_box (-30, -30, 30, 30);
    for (n = ngh->up, y = -0.5; n; n = n->next, y += 1.0) {
        for (i = 0, x = n->skip - 0.5, mask = 1, l = n->line;
             i < n->length; ++i, x += 1.0) {
```

```
            if (*l & mask) { do_card (sc, x, y); ++t; }
            if (mask == 0x80) { mask = 1; ++l; }
            else mask <<= 1;
        }
    }
                                                                    410
    for (n = ngh->down, y = -1.5; n; n = n->next, y -= 1.0) {
        for (i = 0, x = n->skip - 0.5, mask = 1, l = n->line;
             i < n->length; ++i, x += 1.0) {
            if (*l & mask) { do_card (sc, x, y); ++t; }
            if (mask == 0x80) { mask = 1; ++l; }
            else mask <<= 1;
        }
    }

    sc->Set_Color (RED);                                            420
    for (x = -19.5; x < 20; x += 1) sc->draw_line (x, -20, x, 20);
    for (y = -19.5; y < 20; y += 1) sc->draw_line (-20, y, 20, y);

    sc->Set_Color (BLACK);
    sc->draw_point (ngh->xoff, ngh->yoff, 1);
    sc->draw_ellipse (r_h, r_v, ngh->xoff, ngh->yoff);

    sc->end_draw();
}
                                                                    430
int do_ellipse_event (X_device *g)
{
    int ev, button = -1, bm = 0, km = 0, key = -1;

    while ((ev = g->get_event()) != NO_EVENT) {
        switch (ev) {
        case EXPOSE : break;
        case MOUSE : g->ev_mouse (button, bm); /* g->ev_point (x, y); */ break;
        case KEY : g->ev_keyboard (key, km); break;
        case RESIZE : break;                                        440
        default : printf ("Main Unknown Event.\n"); break;
        }
    }
    button |= bm;

    switch (button) {
        case -1 : break;
        case LEFT_BUTTON :
        case MIDDLE_BUTTON : return 1; break;
        case RIGHT_BUTTON : return -1; break; // stop/start display  450
    }

    if (key == 'Q') return -1;              // Exit
    if (key >= 0) return 1;
```

```
    usleep (100000L);
    return 0;
}

main (int argc, char *argv[])                                          460
{
    sc = new X_device (-20.0, -20.0, 20.0, 20.0,
                       "There goes the Neighborhood", 800, 800, 0, 0);
    float a, b, c;
    hood_dist *hoods;
    const int nhoods = 1024;
    int i;
    neighbors *n;

    a = atof (argv[1]);                                                470
    c = atof (argv[2]);
    hoods = make_hoods (nhoods, a, c, 1.0, 0);

    for (;;) {
        n = (*hoods)();
        do_ellipse (sc, n, a, c);
        while (! (i = do_ellipse_event ((X_device *)sc))) ;
        if (i < 0) break;
    }
}                                                                      480

#endif
```

A.5 threshold.cc

The code responsible for the calculations of threshold described in section 2.4, neighborhood radius described in section 2.5, and power decay described in section 2.7 is in the file *threshold.cc.*

```
#include <stdio.h>
#include "Element.h"
#include "logfile.h"
#include "Lattice.h"

float rads [MAXSPEED + 1];
int thresh [MAXSPEED + 1];

extern float time_step;

// The neighborhood radius is specified by the speed and diffusion constant
// It also takes the lattice spacing (scale) and returns the neighborhood
// radius in (float) multiples of the lattice spacing.
float calc_radius (double D, int speed, float scale)
{
    return sqrt (6 * D * time_step + speed * speed * time_step * time_step) /
        scale;
}

// this is not meaningful unless nspeed ≡ trigger speed
void calc_radii (double D, int nspeed, float scale)
{
    int speed;

    for (speed = 1; speed <= MAXSPEED; ++speed)
        rads[speed] = calc_radius (D, nspeed, scale);
}

// This function is based on the analytic solution to the threshold for a
// decaying source.  This is not used in the simulator (4/24/98)
float calc_thresh (int speed, float gamma = 0)
{
    double kh, alpha, ch;

    ch = (speed * time_step) / (rads[speed] * lattice->get_scale());

    alpha = 2 * acos (ch);

    kh = alpha * (1 + gamma) -
        (2 * sqrt (1 - ch * ch) / ch) *
        (2 * gamma / 3 + (ch * ch) * (1 + gamma / 3));
```

```
    return  rads[speed] * rads[speed] * kh / 2;
}

// This function calculates the threshold at a given speed given the knowledge
// of the power of an EXCITING element at each time step
float calc_exact_thresh (int speed, float gamma = 0)
{
    int n, i;                                                          50
    double area, last_area = 0, thresh = 0, alpha;
    double power;

    n = (int)floor ((lattice->get_scale() * rads[speed]) / (speed * time_step));
#ifdef TEST
    printf ("calculating threshold for c = %d.  Steps: %d.\n", speed, n);
#endif
    for (i = n;  i > 0;  --i) {
        alpha = 2 * acos (i * speed * time_step /
                          (rads[speed] * lattice->get_scale()));       60
        area = alpha - sin (alpha);  area -= last_area;
        power = 1 - (i - 0.5) * gamma;
        thresh += power * area;
#ifdef TEST
        printf ("power: %f, threshold: %f.\n", power, thresh);
#endif
        last_area = area;
    }

    return rads[speed] * rads[speed] * thresh / 2;                     70
}

// Convert the treshold in power to the fractional area of the neighborhood
static double area_thresh (int speed, float aniso)
{
    return double (thresh[speed] / (1 << 8)) /
        (rads[speed] * rads[speed] * 3.14159 * aniso);
}

// Convert the trheshold in power to the number of fully-excited elements  80
static int num_thresh (int speed)
{
    return int (thresh[speed] / excit_power[active_time-1] + 0.5);
}

int active_time;
unsigned int excit_power[256] = { 1 << 8 };

#define POWER_SCALE(p) int ((1 << 8) * (p) + 0.5)
                                                                       90
int scale_power (float p) { return POWER_SCALE (p); }
```

```
// Calculate the decay profile of power sourced by an element after excitation.
int set_active_power (float gamma, int act_ticks = 0)
{
    double ch_min;
    double t;
    int i;

    ch_min = min_speed  * time_step / (rads[min_speed] * lattice->get_scale());
    if (! act_ticks && gamma) act_ticks = int (1 / gamma + 0.5);

    if (act_ticks < ((int)((1 - ch_min) / ch_min) + 1) && gamma == 0) {
        fprintf (stdout, "Min speed: %d.  Active time (%d) ",
                 min_speed, act_ticks);
        act_ticks = int ((1 - ch_min) / ch_min + 1);
        fprintf (stdout, "reset to %d (power = %f).\n",
                 act_ticks, 1 - gamma * (act_ticks - 0.5));
    }
    printf ("gamma: %f, act_ticks: %d\n", gamma, act_ticks);
    for (i = act_ticks, t = 0.5;  --i >= 0;  t += 1)
        excit_power[i] = POWER_SCALE (1 - gamma * t);

    return act_ticks;
}

/*
  The calculation of γ is based on the assumption that the recovery
  process will lower the plane-wave propagation speed by lowering the power
  delivered from the active wavefront.  Theh calculation assumes a
  "correction" from trigger-wave (no recovery) speed to target-wave (with
  recovery) speed.  The trigger-wave and target-wave thresholds are
  calculated.  The target-wave threshold is taken to be the area of the
  neighborhood covered by the actual plane wave.  The trigger-wave threshold is
  taken to be the normalized power delivered to the neighborhood.  Thus, the
  actual wave covers a larger area of the neighborhood but, since it delivers
  less than full power (because of recovery) it is as though it had
  covered a smaller area (the area given by the trigger wave threshold) but at
  full power.

  γ, therefore, represents the decay from full power to the power which
  would convert a target-wave threshold area to a (full-power) trigger-wave
  threshold power.  Both trigger and target speed are assumed large enough so
  that the waveback does not enter the neighborhood, so the reduction in power
  is during the first time step.  Since recovery is assumed linear, the power
  power during a time step is the average of the beginning and end, so $P(T)$,
  for T = 0, 1, 2, ... is 1 − γ(T + 1/2), and the reduction in
  power for the first time step is 1 − γ/2.  From the above
  argument, 1 − γ/2, the power delivered during the first time
  step,  is the ratio of the trigger threshold to the target threshold.
 */
float set_gamma (int trigger_speed, int target_speed)
{
```

```
    float calc_exact_thresh (int speed, float gamma = 0);
    double trigger_thresh, target_thresh;

    trigger_thresh = calc_exact_thresh (trigger_speed, 0);
    target_thresh = calc_exact_thresh (target_speed, 0);
    return 2 * (target_thresh - trigger_thresh) / target_thresh;
}

float set_exact_thresh (int trigger_speed, int target_speed)
{
    float calc_exact_thresh (int speed, float gamma = 0);
    float gamma;
    int speed;

    gamma = set_gamma (trigger_speed, target_speed);

    for (speed = 1;  speed <= MAXSPEED;  ++speed)
        thresh[speed] = POWER_SCALE (calc_exact_thresh (speed, gamma));

    return gamma;
}

// calculates recovery-corrected thresholds and returns recovery rate constant
// (γ). Based on the analytic solution to the linear recovery
// problem.
float set_thresh (int trigger_speed, int target_speed)
{
    float calc_thresh (int speed, float gamma = 0);
    double trigger_thresh, ch_targ, alpha_targ;
    double kh_trig, gamma;
    int speed;

    // first get the trigger case
    trigger_thresh = calc_thresh (trigger_speed, 0);
    printf ("trigger_threshold: %f.\n", trigger_thresh);
    kh_trig = 2 * trigger_thresh / (rads[trigger_speed] * rads[trigger_speed]);
    // then get the gamma correction
    ch_targ = target_speed * time_step /
        (rads[trigger_speed] * lattice->get_scale());
    alpha_targ = 2 * acos (ch_targ);
    printf ("kh_trig: %f, alpha_targ: %f, ch_targ: %f\n",
            kh_trig, alpha_targ, ch_targ);
    gamma = ((kh_trig - alpha_targ) +
             2 * ch_targ * sqrt (1 - ch_targ*ch_targ)) /
            (alpha_targ - (4 * sqrt (1 - ch_targ*ch_targ)) / (3 * ch_targ) -
                          2 * ch_targ * sqrt (1 - ch_targ*ch_targ) / 3);

    // This should be redundant, but it ensures a clean zero
    if (trigger_speed == target_speed) gamma = 0;

    for (speed = 1;  speed <= MAXSPEED;  ++speed)
```

```
        thresh[speed] = POWER_SCALE (calc_thresh (speed, gamma));

    return gamma;
}

void print_propagate_params (void)                                          200
{
    char buf[132];
    int i;

    printf ("Min speed: %d cm/sec, neighborhood radius: %.2f, "
            "threshold: %d:%.3g.\n",
            min_speed, rads[min_speed],
            num_thresh (min_speed), area_thresh (min_speed, 1.0));
    printf ("Base speed: %d cm/sec, neighborhood radius: %.2f, "
            "threshold: %d:%.3g.\n",                                        210
            def_speed, rads[def_speed],
            num_thresh (def_speed), area_thresh (def_speed, 1.0));

    printf ("active power:"); buf[0] = '\0';
    for (i = active_time;  --i >= 0;  ) {
        printf (" %.3f", (float)excit_power[i] / (1 << 8));
        sprintf (buf, "%s%s%.3f",
                 buf, buf[0] ? " " : "", (float)excit_power[i] / (1 << 8));
    }
    putchar ('\n');                                                         220
#ifndef TEST
    log_qval ("active", buf);
#endif
}

#ifdef TEST

// Routines to test the code.  Not used in the simulator.

int def_speed, min_speed = 30, max_speed = 60;                              230
float length = 0.01, step = 0.001;

hood_dist *make_hoods (int, float, float, float = 1.0, float = 0.0)
{
    return 0;
}

int main (int argc, char *argv[])
{
    float D = 1.0, gamma;                                                   240
    char buf[132];
    int speed, trig, targ;

    for (;;) {
        printf ("enter trigger, target, speed.\n");
```

```
        gets (buf);
        sscanf (buf, "%d %d %d", &trig, &targ, &speed);
        def_speed = speed;
        calc_radii (D, 1.0, trig);
        printf ("radius is %.2f\n", rads[def_speed]);                    250
        printf ("gamma is: %.2f\n", gamma = set_gamma (trig, targ));
        printf ("threshold is: %.3f\n", calc_exact_thresh (speed, 1, gamma));
    }
    return 0;
}

#endif
```

A.6 Lattice.h

After defining the behavior of individual tissue elements in `class Element` (Sec. A.1), the next important step is to construct the macroscopic tissue structure from the individual tissue elements. The macroscopic structure of the tissue is represented by `class Lattice`, which is defined in the file *Lattice.h*.

```
#ifndef LATTICE_H
#define LATTICE_H
#include <math.h>

#ifndef PI
#define PI 3.14159265359
#endif

/*
  A Lattice is designed to be rectangular with periodic boundary conditions in
  x.  Use of the lattice requires all of the pointers for the individual
  Elements: right, left, front, and back, to be meaningful.  The x-direction
  is traversed through right and left, the y-direction through
  front and back (z-direction would be up and down).  Because
  of the PBCs on x, right and left will always be connected for
  elements inside the lattice.

  Lattice geometries are supported as subclasses of Lattice, all
  designed to be used interchangably.  A Lattice is always initially allocated
  as a cylinder, that is, a rectangular array, with horizontal PBCs (right edge
  linked to left edge and vice versa), and a row of DUMMY elements at the
  top and bottom.  To support other geometries elements are removed from the
  lattice by converting them to DUMMY elements and linking around them.

  Because it is necessary to use all lattice types interchangably, access to
  specific locations on the lattice, and determining the location of lattice
  elements in standardized.  There are three different coordinates systems
  which can be used for describing locations in the lattices.  The most basic
  is the two-dimension array notation, which is assumed whenever location is
  specified by integers.  The location is specified by the pair
  (int x, int y), where x varies most rapidly.  Most routines using this
  coordinate system disregard the buffer rows at the top and bottom, with the
  exception of the index() routine.

  The second location scheme is based on float coordinates (again x and y),
  generally overloading the functions previously described but with float
  arguments.  These coordinates are based on the "physical" (rectangular)
  dimensions specified for the lattice.  These functions are guaranteed to
  return valid (non-DUMMY) lattice elements for any reasonable coordinate.
```

This is accomplished by mapping a non-rectangular lattice onto a "distorted rectangle" by "stretching" the final lattice to make it rectangular. Thus for a spheroidal lattice, the x-metric expands near the poles. The float functions all have a "normalized" form, which is the function name prefixed by **nrm_** *which takes the rectangle to be* $[0,1]$.

The third location scheme is based on float coordinates (ϕ,θ), *and specifies the "lattitude" and "longitude" of a lattice location. This is especially useful in spheroidal geometries for proper mapping of geometrical objects on non-rectangular surfaces. The functions have the prefix* **angle_**, *and take values* $\phi \in [0,2\pi]$, *where* $\phi = 0$ *on the x-axis, and* $\theta \in [0,\pi]$, *where* $\theta = 0$ *at the north pole. Not all angles may be used, depending on lattice geometry.*

Although different types of lattices are designed to have the same interface, it is appropriate to determine the types of coordinates best suited to the particular type of lattice. Normally, angle coordinates are best for spheroid and hemispheroid, rectangular coordinates for cylinder and torus.

In practice, it is often useful to use a fourth scheme: direct calculation of the three-dimensional coordinates of points on the surface. This allows exact specification of geometry based on whether the calculated coordinate matches the conditions for the geometry. This is done in the files GeomObj.h and geomobj.cc.
*/

```
// A lat_line contains length and offset information within the allocated
// cynlindrical lattice to find the active elements on that row for the
// sublattice.
struct lat_line {
    int offset, length;
    lat_line (void) { offset = length = 0; }
    lat_line (int o, int l) : offset (o), length (l) { ; }
};

class Lattice {
protected:
    const int x_size;                  // total allocated horizontal length
    const int y_size;                  // vertical length (without BUFFER)
    const unsigned int lattice_size;   // size without BUFFER
    const unsigned int alloc_size;     // size with BUFFER
    Element *const start_alloc;        // start of allocated memory
    Element *const start_lattice;      // start of usable memory (no BUFFER)
    Element *const end_lattice;        // end of usable memory (no BUFFER)
    Element *const end_alloc;          // end of allocated memory
    const float scale;                 // square "size" of lattice element
    lat_line *rows;                    // table of row lengths
    // Access lattice as 2D array
    // No safety check to allow access of buffer rings
    Element *index (int i, int j) {
        return start_lattice + j * x_size + i;
```

```
    }
    // Access lattice as 2D array with safety check-can't get BUFFER
    Element *offset (int x, int y) const {
        if (x < 0 || y < 0 || x >= x_size || y >= y_size) {
            fprintf (stderr,
                    "Oops: %s, %d.  Lattice element %d, %d out of range.\n",
                    __FILE__, __LINE__, x, y);
            return 0;
        }
        return start_lattice + y * x_size + x;                          100
    }
    // $x, y \in [0, 1]$ mapped onto full height, row length.
    Element *offset (float x, float y) const {
        int tx, ty;
        ty = int (y * (y_size - 1) + 0.5);
        if (ty < 0 || ty >= y_size || x < 0 || x > 1.0) {
            fprintf (stderr,
                    "Oops: %s, %d.  "
                    "Lattice element %.3f, %.3f out of range.\n",
                    __FILE__, __LINE__, x, y);                          110
            return 0;
        }
        if (rows[ty].offset < 0) {
            fprintf (stderr,
                    "Oops: %s, %d.  "
                    "Lattice row at %.3f not in lattice.\n",
                    __FILE__, __LINE__, y);
            return 0;
        }
        tx = rows[ty].offset + int (x * (rows[ty].length - 1) + 0.5);   120
        return start_lattice + ty * x_size + tx;
    }
    // returns the length of a particular row.  It is used, in general, when
    // the lattice slices are being set up to determine how long each slice
    // should be.
    virtual int get_x_length (int y) { return x_size; }
    // sets up appropriate links for particlar lattice geometry
    virtual void setup_links (void);
    // link two adjacent rows of different length together
    void link_rows (int, int, int, int, int, int, class Uniform * = 0);  130
    void link_rows (int, int, int, int, int, int,
                    class Uniform *, class Normal *);
    void setup_slices (void);
public:
    // the BUFFER is the top/bottom region to defines the boundary conditions
#define BUFFER_RING (x_size)
#define ALLOC_SIZE (x_size * y_size + 2 * BUFFER_RING)
    // the lattice size is specified in physical units (cm)
    Lattice (float xlen, float ylen, float sc) :
        x_size (int (xlen / sc)), y_size (int (ylen / sc)),             140
        lattice_size (x_size * y_size),
```

```
    alloc_size  (lattice_size + 2 * BUFFER_RING),
    start_alloc (new Element [ALLOC_SIZE]),
    end_alloc (start_alloc + ALLOC_SIZE),
    start_lattice (start_alloc + BUFFER_RING),
    end_lattice (start_lattice + lattice_size),
    scale (sc)
{
    Element *e;
    for (e = start_alloc;  e < start_lattice;  ++e) e->init_dummy();
    for (e = end_lattice;  e < end_alloc;  ++e) e->init_dummy();
    rows = new lat_line [ y_size ] ;
    setup_links();
}
// the start of the usable (non BUFFER) lattice
Element *const lattice_start (void) const { return start_lattice; }
// with no arguemnts, functional form returns start_lattice
Element *const operator() (void) const { return start_lattice; }
// with integer arguments, 2D index into lattice
Element *operator() (int x, int y) const { return offset (x, y); }
// Map an arbitrary pair onto the lattice (modulo 1)
// Modified in this case to work for spheroid
virtual Element *nrm_map (float x, float y) const = 0;
// Find location of element in lattice on normalized [0,1] mapping
void nrm_location (Element* e, float& x, float& y) const {
    int offset = e - start_lattice, ty;
    ty = offset / x_size;
    x = float (offset % x_size);
    x -= rows[ty].offset;  x /= rows[ty].length;
    y = (float)ty / y_size;
}
// On a general the angle stuff is not very well defined, so map the
// rectangle to the angles
virtual Element *angle_map (float phi, float theta) {
    float x = phi / (2 * PI), y = 1 - theta / PI;
    return nrm_map (x, y);
}
virtual void angle_location (Element *e, float &phi, float &theta) {
    nrm_location (e, phi, theta);
    phi *= 2 * PI; theta = 1 - theta * PI;
}
// Find (distored) physical x, y position of element in lattice array
void location (Element* e, int& x, int& y) const {
    int offset = e - start_lattice;
    x = offset % x_size;   y = offset / x_size;
}
/*
  All float location operators take (return) coordinates in the distored
  rectangle created by the lattice
 */
// find physical x, y position of element in distorted rectangle
void location (Element *e, float& x, float& y) const {
```

```
        nrm_location (e, x, y);
        x *= x_size * scale;   y *= y_size * scale;
    }
    // convert (x, y) location in distorted rectangle to lattice element
    // cyclically map any (x, y) onto distorted rectangle and return element
    Element *map (float x, float y) const {
        x /= x_size * scale;   y /= y_size * scale;
        return nrm_map (x, y);                                          200
    }
    // get (dx, dy) for an element at that position in distorted rectangle
    void size (Element *e, float& x, float& y) const {
        int ty = (e - start_lattice) / x_size;
        x = scale * x_size / rows[ty].length;
        y = scale;
    }
    Element *const lattice_end (void) const { return end_lattice; }
    const unsigned int get_lattice_size (void) const { return lattice_size; }
    const unsigned int get_alloc_size (void) const { return alloc_size; }   210
    Element *const alloc_start (void) const { return start_alloc; }
    void get_dimensions (int& x, int& y) const { x = x_size; y = y_size; }
    const int get_x_len (void) const { return x_size; }
    const int get_y_len (void) const { return y_size; }
    // gets horizontal and vertical semi axis lengths (for cylinder)
    virtual void get_rad (float &h, float &v) const {
        h = x_size * scale / (2 * PI);  v = y_size * scale;
    }
    const float get_scale (void) const { return scale; }
    void bad_elem (Element *, char *);                                  220
    int check_lattice (void);
    virtual float area (void) { return (x_size * scale) * (y_size * scale); }
};

/*
   Each independent lattice geometry is supported as a subclass of Lattice.
   The Cylinder should be considered a "degenerate" subclass, since it is the
   most obvious lattice to create.  (Note that a rectangle, i.e. no horizontal
   PBCx is NOT supported, since there is no mechanism in the code for handling
   horizontally-unconnected elements.)                                  230
 */

class Cylinder : public Lattice {
public:
    Cylinder (float xlen, float ylen, float sc) : Lattice (xlen, ylen, sc) { ;}
    // nrm_map maps x cyclically
    virtual Element *nrm_map (float x, float y) const {
        double temp;
        x = modf (x, &temp);
        if (y < 0 || y > 1) return 0;                                   240
        return offset (x, y);
    }
};
```

```
/*
   A Torus is the second simple lattice geometry, where the PBC are vertical as
   well as horizontal.
 */
class Torus : public Lattice {
  public:                                                              250
    // The constructor also links the top/bottom rows
    Torus (float xlen, float ylen, float sc) : Lattice (xlen, ylen, sc) {
        int i;
        Element *te, *be;

        link_rows (y_size-1, 0, 0, x_size, 0, x_size);
    }
    // mapping is cyclic in x and y
    virtual Element *nrm_map (float x, float y) const {
        double temp;                                                   260
        x = modf (x, &temp); y = modf (y, &temp);
        return offset (x, y);
    }
};

/*
   A Sinusoid is a bizzare lattice used for testing purposes.  It is a
   vase-like struct varying in diameter by a factor of three over
   2½ sinusoidal periods.
 */                                                                    270
class Sinusoid : public Lattice {
    // return the length of a given slice (row)
    int get_x_length (int y, int x_size, int y_size) {
        return round (((2 + cos (3.1416 * 5 * y / y_size)) * x_size) / 3.0);
    }
  public:
    // Again, the construct builds the special geometry
    Sinusoid (float xlen, float ylen, float sc) : Lattice (xlen, ylen, sc) {
        setup_slices();
    }                                                                  280
    // only x is cyclic
    virtual Element *nrm_map (float x, float y) const {
        double temp;
        x = modf (x, &temp);
        if (y < 0 || y > 1) return 0;
        return offset (x, y);
    }
};

/*                                                                     290
   A Hemispheroid is designed for the implementation of a single ventricle.  It
   is open and buffered at y = 0 and closed at y = 1 (normal mapping).  There
   is obviously no zero-size slice–the smallest slice has a length of twice
   the neighborhood radius.
```

```
*/
class Hemispheroid : public Lattice {
    // length of slice is hemishperoid
    int get_x_length (int y) {
        return round (x_size * sqrt (1 - (float)y * y / (y_size * y_size)));
    }
  public:
    // construct sets up geometry
    Hemispheroid (float xlen, float ylen, float sc) :
        Lattice (xlen, ylen, sc) {
        setup_slices();
    }
    // x is cyclically mapped, y is folded across 1
    virtual Element *nrm_map (float x, float y) const {
        double temp;
        x = modf (x, &temp);
        if (y < 0 || y > 2) return 0;
        if (y > 1) {
            y = 2 - y;
            x = modf (x + 0.5, &temp);
        }
        return offset (x, y);
    }
    // A hemispheroid has only a Northern hemisphere
    Element *angle_map (float phi, float theta) {
        float x, y;
        if (theta > PI / 2) return 0;
        x = phi / (2 * PI);   y = 1 - (2 * theta / PI);
        return nrm_map (x, y);
    }
    void angle_location (Element *e, float &phi, float &theta)
    {
        nrm_location (e, phi, theta);
        phi *= 2 * PI;
        theta = (1 - theta) * PI / 2;
    }
    // gets horizontal and vertical semi axis lengths (for cylinder)
    void get_rad (float &h, float &v) const {
        h = x_size * scale / (2 * PI);  v = 2 * y_size * scale / PI;
    }
    // area is less than lattice area.
    float area (void) {
        return (PI / 4) * (x_size * scale) * (y_size * scale);
    }
};

/*
   A Spheroid is symmetric and closed at y = 0 and y = 1.  The same
   considerations apply as for a hemispheroid.  It is is used for atria.
 */
class Spheroid : public Lattice {
```

```
    int get_x_length (int y) {
        float y_off = (float(y) - y_size / 2) / y_size * 2;
        return round (x_size * sqrt (1 - y_off * y_off));
    }

public:
    Spheroid (float xlen, float ylen, float sc) : Lattice (xlen, ylen, sc) {
        setup_slices();
    }
    virtual Element *nrm_map (float x, float y) const {
        double temp;
        x = modf (x, &temp);
        if (y < -1 || y > 2) return 0;
        if (y > 1) {
            y = 2 - y; x = modf (x + 0.5, &temp);
        }
        else if (y < 0) {
            y = -y;   x = modf (x + 0.5, &temp);
        }
        return offset (x, y);
    }
    Element *angle_map (float phi, float theta) {
        float x, y;
        x = phi / (2 * PI);   y = 1 - (theta / PI);
        return nrm_map (x, y);
    }
    void angle_location (Element *e, float &phi, float &theta)
    {
        nrm_location (e, phi, theta);
        phi *= 2 * PI;
        theta = (1 - theta) * PI;
    }
    // gets horizontal and vertical semi axis lengths (for cylinder)
    void get_rad (float &h, float &v) const {
        h = x_size * scale / (2 * PI);  v = y_size * scale / PI;
    }
    float area (void) {
        return (PI / 4) * (x_size * scale) * (y_size * scale);
    }
};

// In general, there is only one active lattice, so I define a single external
// pointer.
extern Lattice *lattice;

#endif
```

A.7 lattice.cc

The routines declared in `class Lattice` are defined in the file *lattice.cc*. This file also contains routines for setting global simulation parameters from those specified in the parameters file, including lattice spacing and time step.

```
#include <stdio.h>
#include <math.h>
#include <SmplStat.h>
#include <SmplHist.h>
#include <iostream.h>
#include <MLCG.h>
#include <Uniform.h>
#include "Element.h"
#include "APD.h"
#include "Lattice.h"                                                    10
#include "Params.h"
#include "logfile.h"
#include "Distribution.h"

void calc_thresh_vals (float, float = 1.0);

Lattice *lattice;
float time_step = 0;
static long rseed = 6735674;
RNG *rand_gen = 0;                                                      20
static int trigger_speed;

void get_params (void);

static float D = 1.0;

int Element::watch_elem (void)
{
    return loc_thresh;
}                                                                       30

/*
  set_params() is called to set the local speed and nominal APD for each
  element.
*/
void set_params (int mean_a, int mean_b, int speed)
{
    int i, j;
    register Element *e;
    int s;                                                              40
```

```
    for (j = 0;  j < lattice->get_y_len();  ++j) {
        for (i = 0;  i < lattice->get_x_len();  ++i) {
            e = (*lattice) (i, j);
            if (e->is_dummy()) continue;
            s = round (mean_b +
                    (j / lattice->get_y_len()) * (mean_a - mean_b));
            e->set_base_ref (s);
            e->set_base_speed (speed);
        }
    }
}

int relref_time = 0;

/*
  set_times() is called by the parameter initialization routine to read
  from the parameter file those parameters that will determine speed and
  recovery.
*/
void set_times (void)
{
    float mean_a = 0, mean_b = 0;
    float tau_a = 0, tau_b = 0;
    int i, p, set_speed = 0;
    void calc_taus (int ta, int tb);
    float match_speeds (int s1, int s2);

    new Param ("mean-a", mean_a); new Param ("mean-b", mean_b);
    new Param ("tau-a", tau_a); new Param ("tau-b", tau_b);
    new Param ("set-speed", set_speed);
    new Param ("relref", relref_time);

    get_params();

    if (! relref_time) relref_time = Dispersion->get_relref();
    def_speed = Dispersion->get_def_speed();
    if (! set_speed) set_speed = def_speed;

    if (tau_a == 0) tau_a = Restitution->get_def_tau() * time_step;
    if (tau_b == 0) tau_b = Restitution->get_def_tau() * time_step;
    calc_taus (round (tau_a / time_step), round (tau_b / time_step));

    if (mean_a == 0) mean_a = Restitution->get_def_apd() * time_step;
    if (mean_b == 0) mean_b = Restitution->get_def_apd() * time_step;

    set_params (round (mean_a / time_step), round (mean_b / time_step),
            set_speed);
    log_val ("set-speed", set_speed);
}
```

```
/*
  make_lattice() is called from the parameters routine to read from the
  parameters file those parameters that determine lattice size and geometry, as
  well as characteristics such as lattice space and diffusion constant.  Seeds
  for the random number generator are also read here to give different
  statistical samples.
*/
void make_lattice (void)                                                    100
{
    float heart_x = 10, heart_y = 5, length;
    int ed_x, ed_y, t_rseed;
    register int i;
    char l_type[20];
    void calc_radii (double D, int nspeed, float scale);


    new Param ("rseed", t_rseed);  // random number seed
    new Param ("heart-x", heart_x);  new Param ("heart-y", heart_y);         110
    new Param ("lattice-space", length);
    new Param ("D", D);              // Diffusion constant
    new Param ("trigger-speed", trigger_speed);
    new Param ("lattice", l_type, "cylinder");

    get_params();

    if (t_rseed) rseed = t_rseed;
    rand_gen = new MLCG (rseed & 0xFF00, rseed >> 16);
                                                                            120
    calc_radii (D, trigger_speed, length);
    switch (l_type[0]) {
        case 'h' :
            lattice = new Hemispheroid (heart_x, heart_y, length); break;
        case 's' :
            lattice = new Spheroid (heart_x, heart_y, length); break;
        case 'c' :
            lattice = new Cylinder (heart_x, heart_y, length); break;
        case 't' :
            lattice = new Torus (heart_x, heart_y, length); break;          130
        case 'w' :
            lattice = new Sinusoid (heart_x, heart_y, length); break;
        default :
            fprintf (stderr, "Unrecognized lattice type: %s.\n", l_type);
    }

    lattice->get_dimensions (ed_x, ed_y);
    printf("Lattice is %.3f cm, %d by %d.  ", length, ed_x, ed_y);
    log_val ("edge-x", ed_x); log_val ("edge-y", ed_y);
    log_val ("dx", length, 3);                                              140
    printf ("Size = %dx%d (%dK)\n",
            lattice->get_alloc_size(), sizeof(Element),
            lattice->get_alloc_size() * sizeof(Element) / (1<<10));
```

```
    printf ("Time step %.2f msec.  ", time_step * 1000);
    log_val ("dt", time_step, 3);
    printf ("Supported speeds %d to %d cm/sec.\n",
            min_speed, max_speed);
}

/*                                                                    150
  set_propagate() is called from the parameters routine to the read from
  the parameters file and process those parameters that control propagation
  speed and default neighborhood.
*/
void set_propagate (void)
{
    int i;
    float t, ch_min, gamma;
    int act_ticks = 0;
    int min_speed = 30;                                               160
    float thresh_override = 0;
    int target_speed = 0;
    Element *e;
    char nbuf[80];
    int set_trigger_ticks (int cmin);
    float set_exact_thresh (int trigger_speed, int target_speed);
    int set_active_power (float gamma, int act_ticks = 0);
    int scale_power (float);
    void print_propagate_params (void);
    void setup_neighborhoods (char *l);                               170

    nbuf[0] = '\0';
    new Param ("target-speed", target_speed);
    // to override calculated threshold (for testing)
    new Param ("threshold", thresh_override);
    new Param ("neighborhoods", nbuf); // default neighborhood

    get_params();

    min_speed = Dispersion->get_min_speed();                          180
    ch_min = min_speed * time_step /
        (rads[min_speed] * lattice->get_scale());

    gamma = set_exact_thresh (trigger_speed, target_speed);
    if (thresh_override) {
        thresh[def_speed] = scale_power (thresh_override);
        printf ("threshold at %d set to %.2f\n", def_speed, thresh_override);
        log_val ("thresh-over", thresh_override, 4);
    }
                                                                      190
    active_time = set_active_power (gamma, act_ticks);

    setup_neighborhoods (nbuf);
```

```
    for (e = (*lattice)(); e < lattice->lattice_end(); ++e)
        if (! e->is_dummy()) e->set_lsp();

    print_propagate_params();
}
                                                                    200
/*
  setup_links() does the initial connection for all lattices.
 */
void Lattice::setup_links (void)
{
    int i, j;

    for (i = 0; i < x_size; ++i) {
        for (j = 0; j < y_size; ++j) {
            if (i > 0) index (i, j)->left = index (i-1, j);                    210
            else index (i, j)->left = index (x_size-1, j);
            if (i < x_size - 1) index (i, j)->right = index (i+1, j);
            else index (i, j)->right = index (0, j);
            index (i,j)->front = index (i, j+1);
            index (i,j)->back = index (i, j-1);
            rows[j].offset = 0; rows[j].length = x_size;
        }
    }
}
                                                                    220
void Lattice::bad_elem (Element *e, char *msg = 0)
{
    int x, y;
    float fx, fy;
    location (e, x, y);
    nrm_location (e, fx, fy);
    printf ("bad element %x, (%d, %d) (%.2f, %.2f).", e, x, y, fx, fy);
    printf ("  (on %d %d).", rows[y].offset, rows[y].offset + rows[y].length);
    if (msg) printf ("  %s.\n", msg);
    else putchar ('\n');                                            230
}

int Lattice::check_lattice (void)
{
    Element *e;
    int i, j;

    for (j = 0;  j < y_size;  ++j) {
        for (i = 0;  i < x_size;  ++i) {
            e = index (i, j);                                       240
            if (e->is_dummy()) continue;
            if (! e->find_front()) bad_elem (e, "Null front");
            if (! e->find_back()) bad_elem (e, "Null back");
            if (! e->find_left()) bad_elem (e, "Null left");
            if (! e->find_right()) bad_elem (e, "Null right");
```

```
            if (j < y_size - 1 && rows[j+1].offset >= 0 &&
                e->find_front()->is_dummy()) bad_elem (e, "Dummy front");
            if (j && rows[j-1].offset >= 0 &&
                e->find_back()->is_dummy()) bad_elem (e, "Dummy back");
            if (e->find_left()->is_dummy()) bad_elem (e, "Dummy left");
            if (e->find_right()->is_dummy()) bad_elem (e, "Dummy right");
        }
    }
    return 0;
}


#define vlink(e1,e2) (e1)->front = (e2), (e2)->back = (e1)

#define hlink(e1,e2) (e1)->left = (e2), (e2)->right = (e1)

inline int next_rand (int start, int len, Uniform *rnd)
{
    return int ((*rnd) () * (len - 1) + start + 0.5);
}

/*
  return a normally distributed random variable with a σ of
  len/4, and a value no smaller than min
  */
int get_normal (int m, double len, int min, Normal *rnd)
{
    float d;

    do {
        d = (*rnd)();
        d = m + d * len / 4;
    } while (d < min);
    return round (d);
}

/*
  get_mod() address the problem of integer-length domains when different
  sized stipes are being matched.  It returns 1 or 0 with approximately the
  frequency given by frac.  Therefore if the length of a domain should be
  3.8, set the length to 3 and call get_mod on frac = 0.8.  80% of the
  time, it will return 1, and 0 the other 20%.  To make sure there aren't too
  many 1's, it keeps track of the total remainder.
  */
static int get_mod (float frac, int& remain, Uniform *unf)
{
    if (remain && (*unf)() < frac) { --remain; return 1; }
    return 0;
}

/*
```

```
  next() and prev() are function to move "forward" and "bacward"
  within a horizontal stripe relative to the current direction.
  */
                                                                    300
inline Element *next (Element *e, int left)
{
    return left ? e->find_right() : e->find_left();
}

inline Element *prev (Element *e, int left)
{
    return left ? e->find_left() : e->find_right();
}
                                                                    310
// This is the simplest form of the routine to vertically link two rows
// together.  If the rows are the smae length, it links them one-to-one.  If
// the rows are different lengths, random links are duplicated in the longer
// row, chosen from the uniform distribution rnd.  If rnd is not
// specified for unequal rows, the routine will die ungracefully.

// A better implementation for unequal rows (which is used in the simulator) is
// below.
void Lattice::link_rows (int l, int h,
                         int high_offset, int low_offset,           320
                         int high_length, int low_length,
                         Uniform *rnd = 0)
{
    static int left = 0;
    Element *eh, *el;  // high element and low element
    int d, m, t;
    int short_len, short_off;
    int short_high = 0; // define high row as smallest y

    left = ! left;                                                  330
    if (low_length < high_length) {
        short_len = low_length;       short_off = low_offset; short_high = 0;
    }
    else {
        short_len = high_length;      short_off = high_offset; short_high = 1;
    }

    if ((d = high_length - low_length) < 0) d = -d; // deficit in elements
    m = next_rand (short_off, short_len, rnd);
    eh = offset (m, h); el = offset (m, l);                         340
    for (t = 0;  d;  --d) {
        m = next_rand (short_off, short_len, rnd);   t += m;
        for ( ;  --m >= 0;  eh = next (eh, left), el = next (el, left))
            vlink (eh, el);
        if (short_high) eh = prev (eh, left); else el = prev (el, left);
        vlink (eh, el);
    }
```

```
    while (t < short_len) {   // not everything linked yet
        el = next (el, left);    eh = next (eh, left);
        vlink (eh, el);
        ++t;
    }

}

/*
  The fundamental problem of building a non-rectangular (non-cylindrical)
  lattice is linking portions of different size.  A lattice is (arbitrarly)
  divied into horizontal stipes which are linked right-to-left (see
  Lattice.h).  Since these stripes are, in general, different lengths, but
  since a wavefront must be able to propagate smoothly between them without
  breaking the symmetry of the lattice, it is necessary to handle the non
  one-to-one nature of the vertical connectivity between the slices by
  staggering some of the vertical links.

  In practice, the size of adjacent slices is compared and the difference,
  d, is the number of links that must be duplicated on the longer row
  (i.e. two adjacent elements in the longer row will point to the same element
  in the shorter row).  These elements are picked at random, but "repel" each
  other.  The short row is divided into d domains, each having (on average)
  a length length / d, where length is the length of the short row
  (see the code for get_mod).  An element to double link is chosen at
  random from each domain from a guassian distribution about the center of the
  domain with a σ of one-half the domain length.  Therefore the
  double-linked elements are approximately uniformly distributed and rarely
  occur close together, but still microscopically random, so no specific
  lattice is imposed on the linkage.
  */
void Lattice::link_rows (int l, int h,
                         int high_offset, int low_offset,
                         int high_length, int low_length,
                         Uniform *unf, Normal *nrm)
{
    static int left = 0;
    Element *eh, *el;  // high element and low element
    int d, m, t, domains, d_len, n, i;
    int short_len, short_off;
    int short_high = 0; // define high row as smallest y
    double frac, dtmp, fd_len;
    int remain;

    left = ! left;        // Each adjacent row is done in opposite directions
    if (low_length < high_length) {
        short_len = low_length;       short_off = low_offset; short_high = 0;
    }
    else {
        short_len = high_length;      short_off = high_offset; short_high = 1;
    }
```

```
    // calculate deficit between lengths of adjacent rows
    if ((d = high_length - low_length) < 0) d = -d;
    m = next_rand (short_off, short_len, unf); // start are random location
    eh = offset (m, h); el = offset (m, l);
    if (d) {
        d_len = short_len / d;          fd_len = (float)short_len / d;
        remain = short_len % d;
        frac = modf ((float)short_len / d, &dtmp);
    } else {
        d_len = 0;
    }
    for (i = 0, m = (d_len + left) / 2;  --d >= 0;
         m += (d_len + get_mod (frac, remain, unf))) {
        n = get_normal (m, fd_len, i, nrm);
        for ( ;  i < n;  eh = next (eh, left), el = next (el, left), ++i)
            vlink (eh, el);
        // left the loop at first pair not linked-back on short row so the
        // last short linked will ALSO link to first long not linked
        if (short_high) eh = prev (eh, left); else el = prev (el, left);
        vlink (eh, el);
        eh = next (eh, left);   el = next (el, left);
    }
    // finish any elements that are not vertically linked
    for ( ;  i < short_len;  eh = next (eh, left), el = next (el, left), ++i)
        vlink (eh, el);
}

// This is the function that cuts a specified lattice geometry into horizontal
// slices and links those slices together into a lattice.
void Lattice::setup_slices (void)
{
    int old_offset, old_length, new_offset, new_length, minrow;
    int y, i, m;
    Uniform rnd (0.0, 1.0, rand_gen);
    Normal nrm (0.0, 1.0, rand_gen);

    // Start with lattice linked normally

    // since the lattice is (arbitrarily) divided into horizontal slices, and
    // the neighborhood scheme is also based on horizontal slices, the
    // neighborhoods defined at the top/bottom would be badly distored if the
    // rows at the top/bottom were less than a neighborhood diameter.
    minrow = int (2 * rads[max_speed] + 0.5);

    for (y = 0;  y < y_size; ++y) {
        if (get_x_length (y) >= minrow) break;
        for (i = 0;  i < x_size;  ++i) index (i, y)->init_dummy();
        rows[y].offset = -1;
    }
```

```
    old_length = old_offset = 0;                                          450
    for ( ; y < y_size; ++y) {    // loop should break at rowmin
        new_length = get_x_length (y);
        if (new_length < minrow) break;
        new_offset = (x_size - new_length) / 2;
        rows[y].offset = new_offset;    rows[y].length = new_length;
        hlink (offset (new_offset, y),
               offset (new_offset + new_length - 1, y));
        for (i = 0; i < new_offset; ++i) index (i, y)->init_dummy();
        for (i = new_offset + new_length; i < x_size; ++i)
            index (i, y)->init_dummy();                                   460
        if (old_length)
            link_rows (y, y-1,
                       old_offset, new_offset, old_length, new_length,
                       &rnd, &nrm);
        old_offset = new_offset;        old_length = new_length;
    }
    new_offset = new_length / 2;
    m = x_size / 2 - 1;
    for (i = 0; i < new_length / 2; ++i) {
        offset (new_offset + i, y-1)->front = offset (m + i, y-1);        470
        offset (m + i, y-1)->front = offset (new_offset + i, y-1);
    }
    for ( ; y < y_size; ++y) {
        rows[y].offset = -1;
        for (i = 0; i < x_size; ++i)
            offset (i, y)->init_dummy();
    }
    check_lattice();
}
```

A.8 APD.h

The file *APD.h* contains the classes which are concerned with specific properties of the action potential. The restitution behavior is defined in the class `Restitution`, and the dispersion behavior is defined in the class `Dispersion`. The calculations associated with these classes are in the file *apd.cc*.

```
#ifndef APD_H
#define APD_H
#include <math.h>
#include <iostream.h>
#include "Queue.h"

// History is not used in the current version of the simulator (4/24/98).
class History {
    // This history is stored as rr,di pairs in a ring buffer of length MAXH.
    // All values are stored as unsigned chars, representing the 8 most
    // signficiant bits of the value.  The number of insignificant bits is
    // given by di_scale and rr_scale.  A value of zero means an unused
    // location.  A value of 255 represnets 255 or greater.
    const int MAXH = 30;          // Number of elements stored
    static int di_scale;          // bits lost storing di's
    static int rr_scale;          // bits lost sotring rr's
    int last_r;                   // time (tics) of last depolarization
    unsigned char cur_pt;         // current location in ring
    unsigned char prev_di[MAXH];
    unsigned char prev_rr[MAXH];
    static float Tau;             // history decay time
    static float h_weight;        // relative weight of history
    static float cur_weight;      // relative weight of current

public:
    History (void) : cur_pt (0), last_r (-1) {
        int i;
        for (i = 0;  i < MAXH;  ++i) prev_di[i] = prev_rr[i] = 0;
    }
    void new_r (int time, int cur_di) {
        int di, rr;
        if ((di = cur_di >> di_scale) > 255) di = 255;
        prev_di[cur_pt] = di;
        if ((rr = (time - last_r) >> rr_scale) > 255) rr = 255;
        prev_rr[cur_pt] = rr;
        last_r = time;
        if (++cur_pt >= MAXH) cur_pt = 0;
    }
    float get_history (int cur_di) {
```

```
        float h, nrm, n;
        int i, t;

        for (i = cur_pt-1, h = nrm = 0, t = 0;  i >= 0  ;  --i) {
            if (prev_rr[i] == 0) break;
            t -= (prev_rr[i] << rr_scale);
            n = exp (t / Tau);
            nrm += n;
            h += (prev_di[i] << di_scale) * nrm;
        }
        for (i = MAXH-1;  i >= cur_pt;  --i) {
            if (prev_rr[i] == 0) break;
            t -= (prev_rr[i] << rr_scale);
            n = exp (t / Tau);
            nrm += n;
            h += (prev_di[i] << di_scale) * nrm;
        }
        if (! nrm) nrm = 1;
        return (h_weight * h) / nrm + cur_weight * cur_di;
    }
};

extern int relref_time; // duration of relative refractory

// Note: the restitution curve used by Joseph Cohen:
//   a(t_r) = T_0(x,y) + T_1(1 - exp(-t_r/T_2),
// Where T_0(x,y) is distributed around 142 msec with a minimum of 71
// msec, T_1 is 272 msec and T_2 is 156 msec.

// The restitution curve is the relationship between action potential and
// previous diastolic interval(s).  The diastolic interval begins immediately
// after the end of the previous action potential.  In the language of the
// simulator, the REFRACTORY period is the action potential duration, and the
// diastolic interval is during the RELREF period and the EXCTIABLE period.
// Current ACTIVE is taken to represent the rapid upstroke and is neither
// action potential nor diastolic interval.
class Restitution_Curve {
protected:
    const int TAU;   // Decay time of exponential (ticks)
    const int MAX_T; // maximum time for function (ticks)
    const int DEF_APD; // resting APD for "healthy" element (ticks)
public:
    Restitution_Curve (float time_step, float apd, float tau) :
        DEF_APD (int (apd / time_step)),
        TAU (int (tau / time_step)),
        MAX_T (100 * int (tau / time_step) * 3) { ; }
    virtual int operator() (int di, int tau, int t0) = 0;
    int get_def_apd (void) { return DEF_APD; }
    int get_def_tau (void) { return TAU; }
    friend ostream& operator<< (ostream& os, Restitution_Curve& d);
};
```

```
// The No_Restitution subclass implements the trival case in which the APD does
// not depend of previous diastolic interval.
class No_Restitution : public Restitution_Curve {
public:
    No_Restitution (float time_step, float apd) :
        Restitution_Curve (time_step, apd, 1.0) { ; }
    int operator() (int di, int tau, int t0) { return DEF_APD; }
};

// The Courtemanche subclass implements the resitution curve determined from
// simulations on Beeler-Reuter model by Courtemanch, et. al.:
//   a(t_r) = 20 + B(t_r)(t_r^5.5/(72^5.5 + t_r^5.5))
//   B(t_r) = 250 - 90 exp(-t_r/145)
//   From PRL 70(14), Courtemanche, Glass, Keener, 93 ( [20])
//   a, t_r in msec.
class Courtemanche : public Restitution_Curve {
    const int DELTA; // strength of exponential (ticks)
    const int OFFSET; // minimum APD (ticks)
    // For speed, the function is converted into an array
    float *r_array;     // function values (fraction of def_apd)

    // The r_function() is used only at initialization to calculate the
    // array values.
    virtual float r_function (float arg);
  public:
    Courtemanche (float time_step);
    // The function operator for this class translates the diastolic interval,
    // in terms of tau to an array index to get the fractional shortening
    // of APD.  It then multiplies the results by the base APD, t0 and
    // returns the next APD.
    int operator() (int di, int tau, int t0) {
        di = di * 100 / tau;
        if (di > MAX_T) return t0;
        return OFFSET + int (r_array[di] * t0 + 0.5);
    }
};

// The Franz subclass implements a fit to the experimental curve measured by
// Franz.  It was not used in the simulations in the thesis (4/24/98).
class Franz : public Restitution_Curve {
    const float CURVE_START;
    const float DELTA; // strength of exponential (ticks)
    const float FERMI_OFF;
    const float FERMI_WID;
    float *t_array;     // function values (fraction of def_apd)
    float *tau_array;

    virtual float tau_function (float arg);
    float t_function (float t);
  public:
```

```
    Franz (float time_step);
    int operator() (int di, int tau, int t0) {
        float r;
        if (di == 0) return 0;
        if (di >= MAX_T) return t0;
        r = t_array[di];
        di = di * 100 / tau;
        if (di >= MAX_T) return t0;
        return int ((r + tau_array[di]) * t0 + 0.5);
    }
    friend ostream& operator<< (ostream& os, Franz& d);
};

// The Dispersion Curve is the relationsihp between an element's excitability
// and the time since the end of that element's previous action potential.  For
// the simulator, the dispersion curve is relevant when an element it is
// reexcited from the RELREF state.  This arbitrary cutoff on an asymptotic
// expression is taken to be at 4τ.
// It implements the function:
//   c(t_r) = 41.7 - 13.5 exp(-(t_r - 37)/18), for t_r > 40
//   c(t_r) = 0 for t_r <= 40
//   where c is in cm/sec and t_r is in msec.
//   Based on PRL, 70(14), Courtemanche, Glass, Keener, 1993.
//   Minimum speed is ≈ 30.3 cm/sec
class Dispersion_Curve {
    const float MIN_RECOVERY = 0.040; // unexcitable time following AP (sec)
    const int RELREF_TIME; // maximum computed time for RELREF (ticks)
#define C_DEF_SPEED 42  // This was what they found from B-R
    const int DEF_SPEED;
    const int MIN_SPEED;
    // The d_function() is similar to the $|r_function()|% discussed in
    // restitution.  It is called only at initialization to fill the array of
    // values.
    int d_function (float t) {
        if (t < MIN_RECOVERY) return 0;
        return int (41.7 -
                    13.5 * exp ((0.037 - t) / 0.018) + 0.5);
    }
    // The function is converted to an array for efficiency.
    unsigned char *d_array;
  public:
    // The RELREF state is on a timer that starts at RELREF_TIME and goes to
    // 0.  Therefore the array to determines the dispersion curve must be
    // filled backward, i.e. t == RELREF_TIME ==> (t == 0).
    Dispersion_Curve (float time_step, int nrm_speed = C_DEF_SPEED) :
        RELREF_TIME (int ((0.037 + 4 * 0.018) / time_step + 0.5)),
        DEF_SPEED (nrm_speed)
    {
        d_array = new unsigned char [ RELREF_TIME ];
        int i;
        float t;
```

```
        for (i =  RELREF_TIME, t = 0;  --i >= 0;  t += time_step)
            d_array[i] = round (d_function (t) * nrm_speed / C_DEF_SPEED);
    }
    unsigned char operator() (int t) { return d_array[t]; }
    float get_min_recovery (void) { return MIN_RECOVERY; }
    int get_relref (void) { return RELREF_TIME; }
    int get_def_speed (void) { return DEF_SPEED; }
    int get_min_speed (void) { return MIN_SPEED; }                          200
    inline friend ostream& operator<< (ostream& os, Dispersion_Curve& d);
};

inline ostream& operator<< (ostream& os, Dispersion_Curve& d)
{
    int i;

    for (i = d.RELREF_TIME;  --i >= 0;  ) os << (int)d(i) << " ";
    os << endl;
    return os;                                                              210
}

// In general there is only one Resitution and Dispersion curve, so I can
// refer to the restitution and dispersion properties globally
extern Restitution_Curve *Restitution;
extern Dispersion_Curve *Dispersion;

struct Beat_time {
    float time;
    Beat_time (float t) : time (t) { ; }                                    220
};

typedef Queue<Beat_time> Beat_queue;

#endif
```

A.9 apd.cc

```
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include "Element.h"
#include "APD.h"
#include "Params.h"
#include "logfile.h"
#include "Lattice.h"

/*                                                                    10
  Implement the relation:
  a(t_r) = 20 + B(t_r)(t_r^5.5/(72^5.5 + t_r^5.5))
  [B(t_r) = 250 - 90 exp(-t_r/145)]
  From PRL 70(14), Courtemanche, Glass, Keener, 1993 [20]
  a, t_r in msec, t_r is the recovery time (time since end of
  action potential)
  */
static float history = 0;

float Courtemanche::r_function (float arg)                            20
{
    if (arg == 0) return 0;
    return (1 - (float (DELTA) / DEF_APD) * exp (-arg)) /
        (pow ((0.072 / TAU) / arg, 5.5) + 1);
}

Courtemanche::Courtemanche (float time_step) :
    Restitution_Curve (time_step, 0.250, 0.145),
    OFFSET (int (0.020 / time_step)),
    DELTA (int (0.090 / time_step))                                   30
{
    int i;
    r_array = new float [ MAX_T ];
    for (i = 0;  i < MAX_T;  ++i)
        r_array[i] = r_function (i / 100.0);
}


// Implement the relation
// a(t_r) = DEF_APD x(1 - exp(-t_r/tau))+                             40
//          DELTA x(log(t_r))/
//              (1 + exp((t_r- FERMI_OFF) / FERMI_WID))
float Franz::tau_function (float arg)
{
    return (1 - exp (-arg - CURVE_START / TAU));
}
```

```
float Franz::t_function (float t)
{
    if (t == 0) return 0;
    return (DELTA / DEF_APD) * log10 (t) /
        (1 + exp ((t - FERMI_OFF) / FERMI_WID));
}

ostream& operator<< (ostream& os, Franz& d)
{
    int i;
    extern float time_step;
    int tau;

    os << "Tau: " << d.TAU << endl;
    for (i = 1;  i < d.TAU * 3;  ++i) {
        os << i * time_step;
        for (tau = 3 * d.TAU / 4;  tau < 4 * d.TAU;  tau *= 2)
            os << " " << d (i, tau, d.DEF_APD) * time_step;
        os << endl;
    }
    os << endl;
    return os;
}

Franz::Franz (float time_step) :
    Restitution_Curve (time_step, 0.270, 0.147),
    CURVE_START (0.235 / time_step),
    DELTA (0.008 / time_step),
    FERMI_OFF ((0.331 - 0.235) / time_step),
    FERMI_WID (0.0067 / time_step)
{
    int i;
    float r;
    cout << "Tau: " << 0.147 << " Time step: " << time_step <<
        " Quot: " << int (0.147 / time_step) <<
        " TAU: " << TAU << " MAX_T: " << MAX_T << endl;
    t_array = new float [ MAX_T ];
    tau_array = new float [ MAX_T ];
    for (i = 0;  i < MAX_T;  ++i)
        tau_array[i] = tau_function (i / 100.0);
    for (i = 0;  i < MAX_T;  ++i)
        if ((t_array[i] = t_function (i)) > .01) break;
    for (i = 0;  i < MAX_T;  ++i)
        if ((t_array[i] = t_function (i)) < .0001) break;
    for (  ;  i < MAX_T;  ++i) t_array[i] = 0;
}

Dispersion_Curve *Dispersion;
Restitution_Curve *Restitution;
```

```
#ifdef HISTORY
static int History::di_scale = 0, History::rr_scale = 2;                    100
static float History::h_weight = 0, History::cur_weight = 1.0;
#endif

static float tau;       // time const for history dependence
float h_weight;         // weight of history dependence
int def_speed = 0;
int min_speed = 17, max_speed = MAXSPEED;

void setup_apd (void)
{                                                                           110
    extern float time_step;
    char restit[20];
    float apd;

    new Param ("time-step", time_step);
    new Param ("Restitution", restit,  "Courtemanche");
    new Param ("R-tau", tau);
    new Param ("H-weight", h_weight);
    new Param ("apd", apd);
    new Param ("def-speed", def_speed);                                     120
    new Param ("c-min", min_speed);

    get_params();

    if (! def_speed)
        Dispersion = new Dispersion_Curve (time_step);
    else
        Dispersion = new Dispersion_Curve (time_step, def_speed);

    //     cout << "Disperson: " << *Dispersion;                             130
    if (tolower (*restit) == 'f') {
        Restitution = new Franz (time_step);
        //     cout << "Restit: " << *(Franz *)Restitution;
    }
    else if (tolower (*restit) == 'c')
        Restitution = new Courtemanche (time_step);
    else if (tolower (*restit) == 'n')
        Restitution = new No_Restitution (time_step, apd);
    else {
        fprintf (stderr, "Unrecognized restitution curve: %s\n", restit);   140
        exit (1);
    }
}

float update_history (int cnt, Beat_queue& bq)
{
    const float tau = 1.0;
    const float cut_off = 10;
    float t, rr;
```

```
    Beat_time *bt;
    extern float step;
    float h = 0, cur_t = cnt * step;

    bq.last();  if (bt = bq.prev()) t = bt->time;
    for (bt = bq.prev();  bt;  bt = bq.prev()) {
        rr = t - bt->time;
        h += rr * exp (-t / tau);
        t = bt->time;
    }
    return history = h;
}

void calc_taus (int ta, int tb)
// This function calculates the time constants for the change of refractory
// period of an Element te with pacing rate. lim refers to the distance between
// the two extreme values of the time constant, ta and tb.
{
    int x, y, max_y = lattice->get_y_len() - 1;
    int tau;

    for (x = 0;  x < lattice->get_x_len();  ++x) {
        for (y = 0;  y < lattice->get_y_len();  ++y) {
            tau = (int)(tb * (float)y/max_y + ta * ((float)max_y - y) / max_y);
            if (! tau) printf ("zero tau: %d %d.\n", x, y);
            (*lattice) (x, y)->set_tau(tau);
        }
    }
}

ostream& operator<< (ostream& os, Restitution_Curve& d)
{
    int i;
    extern float time_step;

    for (i = 1;  i < d.TAU*3;  ++i)
        os << i * time_step << " " <<
            d(i, d.TAU, d.DEF_APD) * time_step << endl;
    os << endl;
    return os;
}
```

A.10 Pace.h

"Paced" beats are the only way of creating activity in the simulator. Therefore, normal, sinus beats, PVCs, and externally applied pacing beats or shocks are all considered "paced." There are several degrees of freedom in paced beats, including timing, size, shape, and strength. Some of the shapes, such as lines and rings, are used only for testing how accurately the simulator reproduces phenomena such as plane wave propagation speed.

There are also mechanisms for forking the simulation before a paced beat to allow comparison of simulation evolution with and without the beat. To simplify using this mechanism, there are classes of paced beats that fork simulations at incremental sizes, strengths, and times. The classes that implement this are all defined in the file *Pace.h.*

Since it is often necessary to determine which Paced beats are actually captured, there is also a `class Capture` which is queued to cause capture detection on that beat.

```
#include "Queue.h"
#include "SortQueue.h"
#include "Distribution.h"
#include "GeomObj.h"
#include <string.h>

/*
  The beat types generally specify the "geometry" of the resulting beat.  A
  "point" is a circle of specified size.  "plane_x" and "line_x" are
  plane waves which propagate in the horizontal direction.  The "plane"
  variety is designed to block in one direction so it forms a continuous loop
  around the lattice.  The "ring_y" travels vertically.  The "obj" type
  excites all elements inside a specific GeomObj (Sec. A.12).
  The "sizes" type is special and forks multiple process on object beats of
  specified sizes.
*/
static char *beat_types[] =
{ "", "point", "obj", "plane_x", "line_x", "ring_y", "rotor", "sizes", 0};
```

```
enum beat_type {
   BT_END = -1, BT_NONE = 0,
   BT_POINT = 1, BT_OBJ, BT_PLANE_X, BT_LINE_X, BT_RING_Y, BT_ROTOR, BT_SIZES
};

enum fork_type { FK_NONE = 0, // execute beat without forking
                 FK_PARENT, // do beat in parent, not child
                 FK_ONEOF, // do next of sequence in child, not parent,
                           // and terminate child at capture determination
                 FK_ALLOF,     // like oneof, but don't stop at capture
                 FK_MULTIPLE, // execute sequence in multiple simultaneous
                           // children
                 FK_NUMTYPES };

class Pace {
protected:
   beat_type btype;
   enum fork_type fork_beat;
   GeomObj *obj;    // A paced beat can be defined to capture all the
                    // tissue in a specified object.
   float x, y;      // the center of a circular beat
   float  cur_t, next_t, last_t;      // scheduling information
   float capture;   // time delay before capture detection
   float thresh;    // fractional threshold to determine capture
   float size;      // Size of wavefront (cm)
   char *id;        // name for reference from other beats
   float strength;  // strength specifies capture of RELREF tissue
   int enabled;     // execution can be conditionally disabled
   Queue<Pace> depend; // list of beat to execute conditionally on capture
public:
   Pace (beat_type bt, float x, float y, GeomObj *o,
         float start = 0, enum fork_type bfork = FK_NONE,
         float end = -1, float sz = 0, float st = 1,
         float cap, float th, char *id = 0) :
      btype (bt), Pace::x (x), Pace::y (y), obj (o), size (sz),
      cur_t (-1), next_t (start), last_t (end), Pace::id (id),
      capture (cap), thresh (th), fork_beat (bfork), enabled (1),
      strength (st)
   { ; }
   // the function operator of the Pace class returns the beat type and
   // location IF there is a beat scheduled at the time it is called.
   virtual beat_type operator() (float t, float& xloc, float& yloc) = 0;
   float when (void) { return next_t; }
   float get_capture (void) { return capture; }
   float get_thresh (void) { return thresh; }
   float get_current (void) { return cur_t; }
   enum fork_type get_fork_type (void) { return fork_beat; }
   // disable/enable execution of the beat
   void disable (void) { enabled = 0; }
   void enable (float t) { enabled = 1; next_t += t; last_t += t; }
   virtual int setup (void) { return 1; }
```

```
    GeomObj &get_obj (void) { return *obj; }
    // called with a beat to be enabled if this beat is captured
    void add_depend (Pace *p) { depend += p; }
    Pace *get_depend (int reset = 0) {
        if (reset) return depend.first(); else return depend.next();
    }
    char *get_id (void) { return id; }
    void get_geom (beat_type& b, float& xloc, float& yloc, float& s) {
        b = btype; xloc = x; yloc = y; s = size;
    }
    float get_strength (void) { return strength; }
    float get_size (void) { return size; }
friend ostream& operator<< (ostream& os, Pace& p);
};

// Pace_Rate is a subclass of Pace which provides for pacing at a constant
// rate.
class Pace_Rate : public Pace {
    float period;
public:
    Pace_Rate (beat_type bt, float x, float y, GeomObj *o,
               float start, enum fork_type f, float end, float s, float st,
               float per, float cap, float th, char *id = 0) :
        Pace (bt, x, y, o, start, f, end, s, st, cap, th, id), period (per)
    { ; }
    beat_type operator() (float t, float& xloc, float& yloc) {
        if (! enabled || next_t < 0) return BT_END;
        if (t < next_t) return BT_NONE;
        xloc = x; yloc = y;
        next_t = t + period;
        if (next_t > last_t && last_t >= 0 || period == 0) next_t = -1;
        cur_t = t;
        return btype;
    }

};

// Pace_Data is a subclass of Pace to allow arbitrary beat timing specified by
// a data file.
class Pace_Data : public Pace {
    float *data;
    int i;
public:
    Pace_Data (beat_type bt, float x, float y, GeomObj *o,
               float start, enum fork_type f, float end, float s, float st,
               char *fname, float cap, float th, float freq, char *id = 0);
    beat_type operator() (float t, float& xloc, float& yloc) {
        if (! enabled || next_t < 0) return BT_END;
        if (t < next_t) return BT_NONE;
        xloc = x; yloc = y;
        while ((next_t = data[i++]) < t  && last_t >= 0  && next_t <= last_t)
```

```
        ;
    if (next_t > last_t && last_t >= 0) next_t = -1;
    cur_t = t;
    return btype;
  }
};

/*
  Pace_Sizes is designed to allow a test of different size beats in
  simultaneous child processes
  */
class Pace_Sizes : public Pace {
  float max_size, inc_size, cur_size; // specification of size sequence
public:
  Pace_Sizes (beat_type bt, float x, float y, GeomObj *obj,
              float start, enum fork_type f, float end,
              float min, float max, float inc, float st,
              float cap, float th, char *id = 0) :
    Pace (bt, x, y, obj, start, FK_MULTIPLE, end, min, st, cap, th, id),
    max_size (max), inc_size (inc), cur_size (min)
  { ; }
  beat_type operator() (float t, float &xloc, float& yloc) {
    if (! enabled || next_t < 0) return BT_END;
    if (t < next_t) return BT_NONE;
    xloc = x; yloc = y;
    next_t = -1;
    cur_t = t;
    return btype;
  }
  int setup (void) {
    if (cur_size > max_size) return 0;
    obj = new  Circle_obj (x, y, cur_size);
    cur_size += inc_size;
    return 1;
  }
  friend ostream& operator<< (ostream &os, Pace_Sizes &p);
};

/*
  Pace_Strengths is designed to allow a test of different strength beats in
  simultaneous child processes
  */
class Pace_Strengths : public Pace {
  float max_strength, inc_strength, cur_strength; // strength sequence
public:
  Pace_Strengths (beat_type bt, float x, float y, GeomObj *obj,
                  float start, enum fork_type f, float end, float size,
                  float min, float max, float inc,
                  float cap, float th, char *id = 0) :
    Pace (bt, x, y, obj, start, FK_MULTIPLE, end, size, min, cap, th, id),
    max_strength (max), inc_strength (inc), cur_strength (min)
```

```
    { ; }
    beat_type operator() (float t, float &xloc, float& yloc) {
        if (! enabled || next_t < 0) return BT_END;
        if (t < next_t) return BT_NONE;
        xloc = x; yloc = y;
        next_t = -1;
        cur_t = t;
        return btype;
    }
    int setup (void) {
        if (cur_strength > max_strength) return 0;
        strength = cur_strength;
        cur_strength += inc_strength;
        return 1;
    }
    friend ostream& operator<< (ostream &os, Pace_Strengths &p);
};

// The Capture class is designed to specified a time and set of parameters to
// determine whether the beat captured. It is queued in a capture queue and
// executed at the appropriate time. In general, the capture criteria is the
// activation of a specified amount of tissue (usually relative to the
// originall beat size) and a specified region (usually centered on the
// orginial beat), at a specified delay from the original beat timing. Other
// events, most notably subsequent beats, can be conditional on capture.
class Capture {
    int beat_id;
    int thresh;
    float b_time, c_time;
    Pace *beat;
    char *buf;
public:
    Capture (int id, int th, float bt, float ct, Pace *b, char *bf) :
        beat_id (id), thresh (th), b_time (bt), c_time (ct),
        beat (b), buf (strdup (bf))
    { ; }
    ~Capture (void) { free (buf); }
    int compare (Capture& o) {
        if (this->c_time < o.c_time) return -1;
        if (this->c_time > o.c_time) return 1;
        if (this->beat < o.beat) return -1;
        if (this->beat > o.beat) return 1;
        return 0;
    }
    float get_check (void) { return c_time; }
    float get_init (void) { return b_time; }
    int get_thresh (void) { return thresh; }
    Pace *get_beat (void) { return beat; }
    char *get_buf (void) { return buf; }
    int get_id (void) { return beat_id; }
};
```

A.11 pace.cc

The file *pace.cc* implements the routines defined in the `Pace` and `Capture` classes, as well as implementing "trips" for determining the location of planewaves at specific times. "Trips" are used only for testing the plane wave behavior of the simulator.

```
#include <stdio.h>
#include <unistd.h>
#include <ctype.h>
#include <iostream.h>
#include <strstream.h>
#include <math.h>
#include <string.h>
#include <SmplStat.h>
#include "Element.h"
#include "Per.h"                                                    10
#include "logfile.h"
#include "Distribution.h"
#include "Lattice.h"
#include "Params.h"
#include "Pace.h"
#include "Process.h"
#include "APD.h"

ostream& operator<< (ostream& os, Pace& p)
{                                                                   20
    os << p.cur_t << " " <<
        beat_types[p.btype] << " " << p.strength << " ";
    switch (p.btype) {
        case BT_POINT :
            os << p.x << " " << p.y << " " << p.size;
            break;
        case BT_RING_Y:
        case BT_ROTOR:  os << p.y; break;
        case BT_PLANE_X : os << p.x; break;
        case BT_OBJ: os << *p.obj; break;                           30
    }
    return os;
}

ostream& operator<< (ostream& os, Pace_Sizes& p)
{
    os << *(Pace *)&p << " Sizes " << p.inc_size << " " << p.max_size;
    return os;
}
```

```
ostream& operator<< (ostream& os, Pace_Strengths& p)
{
    os << *(Pace *)&p << " Strengths " <<
        p.inc_strength << " " << p.max_strength;
    return os;
}

Pace_Data::Pace_Data (beat_type bt, float x, float y, GeomObj *o,
                      float start, enum fork_type f, float end, float s,
                      float st,
                      char *fname, float cap, float th, float freq = 361.3,
                      char *id) :
    Pace (bt, x, y, o, start, f, end, s, st, cap, th, id)
{
    FILE *dfp;
    const int MAXSMP = 100000;
    float dbuf[MAXSMP];
    char buf[256];

    if (! (dfp = fopen (fname, "r"))) {
        fprintf (stderr, "Unable to open HR data file %s for read.\n", fname);
        last_t = 0;
        return;
    }

    for (i = 0;  fgets (buf, 256, dfp);  ++i) dbuf[i] = atof (buf) / freq;
    dbuf[i++] = -1;
    data = new float [i];


    for (i = 0;  dbuf[i] >= 0;  ++i) data[i] = dbuf[i];
    data[i] = -1;

    next_t = data[0];
    i = 1;
}

/*
  The following several functions are to build the pace queue from the
  parameters file.
*/
static Queue<Pace> beats;

static int match_key (char *word, char *keys[])
{
    int i;

    for (i = 0;  keys[i];  ++i) if (strcmp (word, keys[i]) == 0) return i;
    return -1;
}
```

```
static char *nextword (char *w, char *l)
{
    *w = 0;
    if (! l) return 0;
    while (*l && isspace (*l)) ++l;
    for ( ; *l && ! isspace (*l); *w++ = *l++) ; *w = 0;
    if (*l) return l+1;
    else return 0;
}

static Pace *find_beat (char *id)
{
    Pace *p;
    char *b;

    for (p = beats.first(); p; p = beats.next())
        if ((b = p->get_id()) && strcmp (b, id) == 0) return p;
    fprintf (stderr, "No beat with id: `%s'.\n", id);
    return 0;
}

// The newbeat() routine reads the specification for a new paced sequence from
// the parameters file. The parameters line for a paced sequences is done as
// a set of <keyword> <value> pairs. The keywords are specified in the
// pace_params array, and the values are handled by the switch statement in
// the routine.
void newbeat (char *l)
{
    beat_type bt = BT_END;
    char *old_l = l;
    float rate = 0, stdev = 0, start = 0, stop = -1, low = 0, high = 100;
    float xloc = -1, yloc = -1;
    float size = 0.3013, strength = 1, max = -1, inc = -1;
    static int i = 0;
    char *id = 0;
    enum fork_type fork_beat = FK_NONE;
    static char *pace_params[] = {
        "x", "y", "obj", "type", "period", "start", "end",
        "data", "freq", "size", "max", "inc", "strength", "id", "sub",
        "capture", "threshold",
        "fork", "oneof", "allof", "multiple", "sizes", "strengths",
        0 };
    enum PACE_PARAMS { X, Y, OBJ, TYPE, PERIOD, START, END,
                       DATA, FREQ, SIZE, MAX, INC, STRENGTH,
                       ID, SUB, CAPTURE, THRESH,
                       FORK, ONEOF, ALLOF, MULTIPLE, SIZES, STRENGTHS,
                       NO_PARAM };
    char word[64], *save_line = l, *w;
    float freq = 361.3;
    float thresh = 0, capture = 0;
```

```
int type = 0;
char *data = 0;
Pace *p, *start_beat = 0;
GeomObj *get_geom_obj (char *&l), *obj = 0;

for (;;) {
    l = nextword (word, l);
    if (! *word) break;
    switch (i = match_key (word, pace_params)) {                    150
        case X: l = nextword (word, l); xloc = atof (word); break;
        case Y: l = nextword (word, l); yloc = atof (word); break;
        case OBJ : obj = get_geom_obj (l); break;
        case TYPE:
            l = nextword (word, l);
            if (! *word) {
                fprintf (stderr, "Beat type missing.\n"); return;
            }
            if ((bt = (beat_type)match_key (word, beat_types)) == BT_END) {
                fprintf (stderr, "Invalid beat type: %s\n", l); return;  160
            }
            break;
        case PERIOD: l = nextword (word, l); rate = atof (word); break;
        case START: l = nextword (word, l); start = atof (word); break;
        case SUB:
            l = nextword (word, l); start_beat = find_beat (word);
            break;
        case END: l = nextword (word, l); stop = atof (word); break;
        case SIZE: l = nextword (word, l); size = atof (word); break;
        case MAX: l = nextword (word, l); max = atof (word); break;  170
        case INC: l = nextword (word, l); inc = atof (word); break;
        case STRENGTH:
            l = nextword (word, l); strength = atof (word); break;
        case CAPTURE: l = nextword (word, l); capture = atof (word); break;
        case THRESH: l = nextword (word, l); thresh = atof (word); break;
        case DATA:
            l = nextword (word, l);
            data = strcpy (new char [strlen(word)+1], word);
            break;
        case FREQ: l = nextword (word, l); freq = atof (word); break;  180
        case FORK: fork_beat = FK_PARENT; break;
        case ONEOF: fork_beat = FK_ONEOF; break;
        case ALLOF: fork_beat = FK_ALLOF; break;
        case MULTIPLE : fork_beat = FK_MULTIPLE; break;
        case SIZES : type = (int)SIZES; fork_beat = FK_MULTIPLE; break;
        case STRENGTHS :
            type = (int)STRENGTHS; fork_beat = FK_MULTIPLE; break;
        case ID :
            l = nextword (word, l);
            id = strcpy (new char [strlen(word)+1], word); break;    190
        default:
            fprintf (stderr, "Unrecognized pacing parameter: %s\n", l);
```

```
            return;
        }
        if (! *word) {
            fprintf (stderr, "Pacing parameter missing.\n"); return;
        }
    }
    if (bt == BT_END) {
        fprintf (stderr, "No pace type specified.\n"); return;                    200
    }
    else if (bt == BT_POINT && xloc < 0 && yloc < 0) {
        fprintf (stderr, "Location not specified for point pacing.\n");
        return;
    }
    else if (bt == BT_RING_Y && yloc < 0) {
        fprintf (stderr, "y-location not specified for horizontal pace.\n");
        return;
    }
    else if (bt == BT_PLANE_X && xloc < 0) {                                      210
        fprintf (stderr, "x-location not specified for vertical pace.\n");
        return;
    }
    else if (bt == BT_ROTOR && yloc < 0) {
        fprintf (stderr, "y-location not specified for rotor initiation.\n");
        return;
    }
    else if (bt == BT_OBJ && ! obj) {
        fprintf (stderr, "no object specified.\n"); return;
    }                                                                             220
    else if ((type == (int)SIZES || bt == BT_SIZES) &&
            (xloc < 0 || yloc < 0 ||
             size < 0 || max < 0 || inc < 0)) {
        fprintf (stderr, "location/size not specified for sizes.\n"); return;
    }
    else if (type == (int)STRENGTHS && ((! obj && (xloc < 0 || yloc < 0)) ||
            max < 0 || inc < 0)) {
        fprintf (stderr, "location/strengths not specified for strengths.\n");
        return;
    }                                                                             230

    if (data)
        p = new Pace_Data (bt, xloc, yloc, obj, start, fork_beat, stop, size,
                           strength, data, capture, thresh, freq, id);
    else if (type == (int)SIZES || bt == BT_SIZES)
        p = new Pace_Sizes (BT_OBJ, xloc, yloc, obj, start, fork_beat, stop,
                            size, max, inc, strength, capture, thresh, id);
    else if (type == (int)STRENGTHS)
        p = new Pace_Strengths (BT_OBJ, xloc, yloc, obj, start,
                                fork_beat, stop,                                  240
                                size, strength, max, inc, capture, thresh, id);
    else
        p = new Pace_Rate (bt, xloc, yloc, obj, start, fork_beat, stop, size,
```

```
                        strength, rate, capture, thresh, id);
    if (start_beat) { start_beat->add_depend (p); p->disable(); }
    beats += p;
}

// Trips are timing events used to determine the location of a plane-wave type
// beat (ring or line) at a specified time.  It is used for testing propagation
// speed.

per *trips[100];

void newtrip (char *l)
{
    int type, rate, start, stop;
    float xloc, yloc;
    static int i = 0;

    if ((i+1) >= sizeof(trips)/sizeof(trips[0])) {
        fprintf (stderr, "Exceeded maximum number of trips: %d.\n", i+1);
        return;
    }
    sscanf (l, "%i %f %f %i %i %i",
            &type, &xloc, &yloc, &rate, &start, &stop);
    if (strchr (l, '\n')) *strchr (l, '\n') = '\0';
    trips[i++] = new per (type, xloc, yloc, rate, start, stop, start);
    trips[i] = NULL;
}

static float field_strength = 1;

void set_field_strength (float f) { field_strength = f; }

/*
  The pace() method activates an element if it's recovery level is great
  enough to be excited by the current field.  The field is specified by
  field_strength, which is compared to the time of recovery since the end
  of the last action potential.
  */
int Element::pace (void)
{
    if (IS_RESTING()) return activate();
    if (IS_RELREF() && timer < round (field_strength * relref_time))
        return activate();
    return 0;
}

static int do_pace (Element *e) { e->pace(); }

static int do_activate (Element *e)
{
    if (e->is_excitable()) return e->activate();
```

```
    return 0;
}

int init_circle (float x, float y, float size)
// This function is used to initiate a circularly spreading wave.
{                                                                          300
    Element *te;
    Hood_dist *get_hoods (int, float, float, float = 1.0, int = 0, int = 0);

    size /= lattice->get_scale();
    return lattice->map (x, y)->
        traverse_neighborhood (do_pace, get_hoods (1, size, size));
}


                                                                           310
int init_ring (int y)
// This function initiates vertically-travelling plane wave at the point
// specified by y.  The depolarized region extends around y by the radius
// given by the default speed.
{
    register Element *a;
    extern int active_time;
    int r = round (active_time * rads[def_speed] / 2);
    int cnt = 0;
                                                                           320
    if (y + r > lattice->get_y_len()) {
        fprintf (stderr, "Ring extends off lattice: %d\n", y);
        return 0;
    }
    for (a = (*lattice) (0, y-r);  a < (*lattice) (0, y+r);  ++a)
        if (a->is_excitable()) {
            cnt += a->activate();
            a->set_lsp();
        }
    return cnt;                                                            330
}

int init_plane_x (int x)
{
    int y, i;
    Element *e;
    int r = 3 * (int)rads[max_speed];
    void put_ds (void);
    int cnt = 0;
                                                                           340
    for (y = 0;  y < lattice->get_y_len();  ++y) {
        e = (*lattice) (x, y);
        for (i = 2 * r, e = e->find_left();  --i >= 0;  e = e->find_left())
            if (e->is_excitable()) e->make_refractory_edge(4);
    }
```

```
    for (y = 0;  y < lattice->get_y_len();  ++y) {
        e = (*lattice) (x, y);
        if (e->is_excitable()) cnt += e->traverse_neighborhood (do_activate);
    }                                                                          350
    return cnt;
}

int init_line_x (int x)
{
    int y, i;
    Element *e;
    int r = 3 * (int)rads[max_speed];
    void put_ds (void);
    int cnt = 0;                                                               360

    for (y = 0;  y < lattice->get_y_len();  ++y) {
        e = (*lattice) (x, y);
        for (i = r, e = e->find_left();  --i >= 0;  e = e->find_left()) {
            if (e->is_excitable()) cnt += e->activate();
        }
    }
    return cnt;
}
                                                                               370
int init_rotor (int yc)
{
    int y, i, x = lattice->get_x_len() / 2;
    Element *e;
    int r = (int)rads[max_speed];
    void put_ds (void);
    int cnt = 0;

    for (y = yc;  y < lattice->get_y_len();  ++y) {
        e = (*lattice) (x, y);                                                 380
        for (i = 2 * r, e = e->find_left();  --i >= 0;  e = e->find_left()) {
            if (e->is_excitable()) {
                e->make_refractory_edge();
            }
        }
    }

    for (y = yc;  y < lattice->get_y_len();  ++y) {
        e = (*lattice) (x, y);
        if (e->is_excitable()) cnt += e->traverse_neighborhood (do_activate);  390
    }
    return cnt;
}

// The capture queue is a list of all capture detects not yet executed
static SortQueue<Capture> capture;
```

```
void log_return (Sim_info *c)
{
    log_val ("child-cnt", c->end_cnt - c->start_cnt);                    400
    log_val ("child-frames", c->end_frame - c->start_frame);
}

// when check_capture is called with a Sim_info struct and a Pace struct, it
// determines whether the Pace event was capture in the child process that
// returned the Sim_info.
int check_capture (Sim_info *c, Pace *b)
{
    extern float time_step;
    return c->last_capt != sim_info_struct->last_capt;                   410
}

// do_beats() is called on each iteration to execute any queued pace events for
// that iteration.  It returns -1 when there are no beats pending, otherwise
// the number of Pace events executed on this interation.
int do_beats (int cnt)
{
    Pace *b, *db;
    int done = 0, active = 0;
    void pace_spike (int);                                               420
    extern float time_step;
    float length = lattice->get_scale();
    float xloc, yloc, size = 0;
    beat_type bt;
    char buf[128];
    ostrstream sbuf (buf, sizeof(buf));
    static int beat_id = 0;          // So beats can be distinguished
    int st;
    int old_maxit, set_maxit (int);
    Sim_info *child_info = 0, *fork_sim (void);                          430

    for (beats.reset();  b = beats.next();  ) {
        bt = (*b) (cnt * time_step, xloc, yloc);
        if (bt == BT_END) continue; // beat has expired
        if (bt == BT_NONE) {++active; continue; } // beat still pending
        switch (b->get_fork_type()) {
            case FK_NONE : break; // execute the beat normally
            case FK_PARENT : // execute beat only if parent
                // fork_sim() returns true if parent
                if (! (child_info = fork_sim())) { b->disable(); continue; }   440
                log_return (child_info);
                break;
            case FK_ONEOF : // each child: one of the beats in the sequence
            case FK_ALLOF :
                if (b->get_capture() > 0)
                    old_maxit = set_maxit (cnt + int (b->get_capture() /
                                                      time_step + 2));
```

```
            else old_maxit = 0;
            if (child_info = fork_sim()) {
                if (old_maxit) set_maxit (old_maxit);
                ++active; log_return (child_info);
                if (b->get_capture() > 0 &&
                    check_capture (child_info, b)) {
                    if (b->get_fork_type() == FK_ONEOF) {
                        log_val ("oneof", cnt);
                        b->disable();
                    }
                    while (db = b->get_depend())
                        db->enable (cnt * time_step);
                }
                continue;
            }
            b->disable();
            break;
        case FK_MULTIPLE : // a separate child for each case
            while (st = b->setup()) {
                if (! (child_info = fork_sim())) break;
                log_return (child_info);
            }
            if (! st) continue; else break;
    }
    ++active;
    ++done;
    ++beat_id;
    set_field_strength (b->get_strength());
    sbuf << beat_id << "; " << *b << ends;
    switch (bt) {
        case BT_POINT :
            st = init_circle (xloc, yloc, b->get_size());
            pace_spike (cnt);
            break;
        case BT_PLANE_X :
            st = init_plane_x (round (xloc / length));
            break;
        case BT_LINE_X :
            st = init_line_x (round (xloc / length));
            break;
        case BT_RING_Y :
            st = init_ring (round (yloc / length));
            break;
        case BT_ROTOR :
            st = init_rotor (round (yloc / length));
            break;
        case BT_OBJ :
            b->get_obj() ((void (*) (Element *))do_pace);
            break;
    }
    capture += new Capture (beat_id, int (b->get_thresh() * st),
```

```
                        cnt * time_step,
                        cnt * time_step + b->get_capture(),
                        b, buf);
    }
    if (done) return done;
    if (active) return 0;
    else return -1;
}

static int num_active;
static void check_active (Element *e)
{
    if (e->IS_ACTIVE()) ++num_active;
}

static int last_capture_id;

Pace *check_capture (int cnt, int active)
{
    extern float time_step;
    float x, y, size, dt, dr;
    GeomObj *obj;
    beat_type bt;
    Capture *tc, *rc;
    Pace *last_bt = 0;
    char buf[128];

    for (tc = capture.first(); tc; ) {
        if (cnt * time_step <= tc->get_check()) {
            tc = capture.next(); continue;
        }
        tc->get_beat()->get_geom (bt, x, y, size);
        if (bt == BT_POINT) {
            dt = cnt * time_step - tc->get_init();
            dr = def_speed * dt;
            obj = new Circle_obj (x, y, dr + size);
            num_active = 0;
            (*obj) (check_active);
            delete obj;
            sprintf (buf, "%s %d %d", tc->get_buf(), cnt, active);
            if (num_active) {
                log_item ("capture", buf); flush_log();
                last_bt = tc->get_beat();
                sim_info_struct->last_capt = last_capture_id = tc->get_id();
                printf ("capture"); fflush (stdout);
            }
            else {
                log_item ("non-capture", buf); flush_log();
                printf ("non-capture"); fflush (stdout);
            }
        }
```

```
        else if (bt == BT_OBJ &&
                 tc->get_beat()->get_obj().get_type() == 'c') {
            dt = cnt * time_step - tc->get_init();
            dr = def_speed * dt;
            obj = &tc->get_beat()->get_obj();
            ((Circle_obj *)obj)->get_circle (x, y, size);
            obj = new Circle_obj (x, y, dr + size);
            num_active = 0;
            (*obj) (check_active);
            sprintf (buf, "%s %d %d", tc->get_buf(), cnt, active);
            delete obj;
            if (num_active) {
                log_item ("capture", buf); flush_log();
                last_bt = tc->get_beat();
                sim_info_struct->last_capt = last_capture_id = tc->get_id();
                printf ("capture"); fflush (stdout);
            }
            else {
                log_item ("non-capture", buf); flush_log();
                printf ("non-capture"); fflush (stdout);
            }
        }
        else if (active > tc->get_thresh()) { // captured
            sprintf (buf, "%s %d %d", tc->get_buf(), cnt, active);
            sim_info_struct->last_capt = last_capture_id = tc->get_id();
            log_item ("capture", buf); flush_log();
            //      printf ("captured: %s\n", buf);
            last_bt = tc->get_beat();
        }
        else {
            sprintf (buf, "%s %d %d", tc->get_buf(), cnt, active);
            log_item ("non-capture", buf);  flush_log();
            //      printf ("not captured: %s\n", buf);
        }
        rc = tc;
        tc = capture.next();
        capture -= rc;
    }
    return last_bt;
}

int do_trips (int cnt)
{
    per **t;
    int active = 0, done = 0;
    int xi, yi;
    SampleStatistic front_loc;
    char buf[512];
    extern float time_step;
    float length = lattice->get_scale();
    const float trip_thresh = 0.3;
```

```
for (t = trips;  *t;  ++t) {
    register Element *e;
    int n = 0, a = 0;
    int x = round ((*t)->xloc / length);
    int y = round ((*t)->yloc / length);
    if ((*t)->next >= 0 && ++active && cnt >= (*t)->next) {
        ++done;
        switch ((*t)->type) {
            // ytrip                                                    610
            case 1 :
                for (e = (*lattice) (0, y);  e < (*lattice) (0, y+1); ++e) {
                    if (e->is_dummy()) continue;
                    ++n;
                    if (e->state_is() == ACTIVE) ++a;
                }
                if (a >= n * trip_thresh) {
                    printf ("\t\ttrip at y = %.2f: time = %.3f.\n",
                            (*t)->yloc, cnt * time_step);
                    sprintf (buf, "%.2f %d", (*t)->yloc, cnt);          620
                    log_item ("y-trip", buf);
                    (*t)->next =
                        cnt + round (60.0 / ((*t)->rate * time_step));
                    if ((*t)->next > (*t)->stop) (*t)->next = -1;
                }
                break;
                // xtrip
            case 2 :
                for (y = 0;  y < lattice->get_y_len();  ++y) {
                    e = (*lattice) (x, y);                              630
                    if (e->is_dummy()) continue;
                    ++n;
                    if (e->state_is() == ACTIVE) ++a;
                }
                if (a >= n * trip_thresh) {
                    printf ("\t\ttrip at x = %.2f: time = %.3f.\n",
                            (*t)->xloc, cnt * time_step);
                    sprintf (buf, "%.2f %d", (*t)->xloc, cnt);
                    log_item ("x-trip", buf);
                    (*t)->next =                                        640
                        cnt + round (60.0 / ((*t)->rate * time_step));
                    if ((*t)->next > (*t)->stop) (*t)->next = -1;
                }
                break;
            case 3 :        // time trip, y
                front_loc.reset();
                for (xi = 0;  xi < lattice->get_x_len();  ++xi)
                    for (yi = lattice->get_y_len();  --yi >= 0;  ) {
                        float xoff, yoff;
                        e = (*lattice)(xi, yi);                         650
                        if (e->IS_ACTIVE()) {
```

```
                        e->get_center_offset (xoff, yoff);
                        front_loc += (yi + yoff) * length;
                        break;
                    }
                }
            if (front_loc.samples()) {
                sprintf (buf, "%d %.4f %.4f %d",
                        cnt, front_loc.mean(), front_loc.stdDev(),
                        front_loc.samples());
                log_item ("t-trip-y", buf);
                printf ("\t\tt-trip-y at time = %.3f, "
                        "location %.3f+/-%.4f. %d of %d\n",
                        cnt * time_step,
                        front_loc.mean(), front_loc.stdDev(),
                        front_loc.samples(), lattice->get_x_len());
            }
            (*t)->next = cnt + round (60.0 / ((*t)->rate * time_step));
            if ((*t)->next > (*t)->stop) (*t)->next = -1;
        }
    }
}
if (done) return done;
if (active) return 0;
else return -1;
}

void setup_events (void)
{
    new Param ("beat", newbeat);
    new Param ("trip", newtrip);

    get_params();
}
```

A.12 GeomObj.h

The file *GeomObj.h* defines classes that allow specification of geometric objects on the lattice. In the final version of the simulator, these objects were always specified in three-space and elements were considered part of the specified objects when their three-dimensional coordinates were inside the specified object. The types of objects implemented were globals, which contained all elements; circles, which were the intersections of spheres with the lattice; and "splashes," which were radially-symmetric objects with properties that depend on radius.

```
#ifndef GEOMOBJ_H
#define GEOMOBJ_H
#include <iostream.h>

/*
  Most geometrical objects are defined in terms of their extent in three-space.
  To do this, the three-space coordinates of all lattice elements are
  calculated and stored in the Coords class pointed to by the global
  coords.  The Point_3D class holds an individual three-space point.
 */

class Point_3D {
public:
    float x, y, z;
    float distance (Point_3D& p2) {
        float t, d;
        t = x - p2.x; d = t * t;
        t = y - p2.y; d += t * t;
        t = z - p2.z; d += t * t;
        return sqrt (d);
    }
};

class Coords {
    Point_3D *lattice_coords;
public:
    Coords (void);
    // The three-space coordinates are indexed parallel to the Element array
    // stored in lattice
    Point_3D *operator() (Element *e) {
        return &lattice_coords[e - lattice->lattice_start()];
    }
};
```

```
// coords class will be initialized by the first routine that needs it,
// which will either be a GeomObj or a SplashObj.
extern Coords *coords;

/*
  A GeomObj as a general class specifying a group of lattice points by
  their location in space.  Most objects will have a least a center point,
  which is a Point_3D.  The interface provides two function operators, one
  which takes an individual element and determines whether that element is
  inside the object, and one that takes a function (taking an element) and
  calls that function for each element inside the object.  There is also a
  general area() function which returns the area of the object (in
  cm^2), and a get_type() which returns a one-letter object type.
 */
class GeomObj {
protected:
    int init;
    Point_3D *center;
public:
    GeomObj (void) : init (0) { ; }
    virtual void setup (float x, float y) {
        if (! coords) coords = new Coords;
        center = (*coords) (lattice->map (x, y));
        init = 1;
    }
    // The function operator of a GeomObj, when called with and Element *, will
    // return true if the element is inside the GeomObj.
    virtual int operator() (Element *) = 0;
    // When the function operator is called with a function (taking an
    // Element *), it will call the function on that element if the element is
    // inside the GeomObj
    virtual void operator() (void (*fn) (class Element *));
    virtual void mbb (float& xmin, float& ymin, float& xmax, float& ymax) {
        return;
    }
    // The type of the GeomObj (single-character code)
    virtual int get_type (void) = 0;
    // The area (projected on the lattice) of the of the GeomObj
    virtual float area (void) { return 0; }
friend ostream& operator<< (ostream& os, GeomObj& p);
};

/*
  A Global_obj contains all points in the lattice
 */
class Global_obj : public GeomObj {
public:
    Global_obj (void) { ; }
    int operator() (Element *e) { return 1; }
    void mbb (float& xmin, float& ymin, float& xmax, float& ymax) { ; }
```

```
    int get_type (void) { return 'G'; }
    float area (void) { return lattice->area(); }
};

/*
  A Circle_obj is actually all the points contained in the three-space sphere
  defined by the specified center and radius.
 */
class Circle_obj : public GeomObj {
    float cx, cy, r;
public:
    Circle_obj (float x, float y, float rad) : cx (x), cy (y), r (rad) { ; }
    virtual int operator() (Element *e) {
        if (! init) setup (cx, cy);
        Point_3D *p = (*coords) (e);
        return center->distance (*p) < r;
    }
    void mbb (float& xmin, float& ymin, float& xmax, float& ymax) {
        xmin = cx - r; ymin = cy - r; xmax = cx + r; ymax = cy + r;
    }
    int get_type (void) { return 'c'; }
    // Return the original circle specification
    void get_circle (float &xc, float &yc, float &rad) {
        xc = cx; yc = cy; rad = r;
    }
    virtual float area (void) { return PI * r * r; }
friend ostream& operator<< (ostream& os, Circle_obj& c);
};

class Rect_obj : public GeomObj {
    float cx, cy, length, height, half_l, half_h, angle;
public:
    Rect_obj (float x, float y, float l, float h, float dir = 0) :
            cx (x), cy (y), length (l), height (h),
            angle (dir) { ; }
    int operator() (Element *e);
    void mbb (float& llx, float& lly, float& urx, float& ury) {
        llx = cx - half_l;      urx = cx + half_l;
        lly = cy - half_h;      ury = cy + half_h;
    }
    int get_type (void) { return 'r'; }
    float area (void) { return length * height; }
};

// A Splash_obj is designed to model a continuous transition between 1 and
// 0 as a function of radius.  It is set up to do infarcts with the level of
// infarction being total for some radius r_min, the dropping according to
// some function of radius, until it is baseline for radii greater than
// r_max.
class Splash_obj : public GeomObj {
protected:
```

```
    float cx, cy, r_min, r_max;
    float (*f_r) (float r);
public:
    Splash_obj (float x, float y, float min, float max, float (*f) (float)) :
        cx (x), cy (y), r_min (min), r_max (max), f_r (f) { ; }
    float rad (Element *e) { return center->distance (*(*coords) (e)); }
    int operator() (Element *e, float& level) {
        if (! init) setup (cx, cy);
        float r = rad (e);
        if (r < r_min) return 1;
        if (r > r_max) return 0;
        level = (*f_r) (1 - (r - r_min) / (r_max - r_min));
        return 1;
    }
    int operator() (Element *e) {
        float level;
        return operator() (e, level);
    }
    int get_type (void) { return 's'; }
    virtual float area (void) { return PI * r_min * r_min; }
friend ostream& operator<< (ostream& os, Splash_obj& s);
};


#endif
```

A.13 geomobj.cc

```
#include <stdio.h>
#include <ctype.h>
#include <math.h>
#include "Element.h"
#include "Lattice.h"
#include "GeomObj.h"
#include <iomanip.h>

char *get_f (float &f, char *l)
{
    char word[32], *w;

    while (*l && isspace (*l)) ++l;
    for  (w = word;  *l && ! isspace (*l);  *w++ = *l++) ; *w = '\0';
    f = atof (word);
    return l;
}

// A routine to parse a GeomObj specification from the parameters file
GeomObj *get_geom_obj (char *&l)
{
    GeomObj *g;
    char word[32], *w, *line = l;

    for (w = word;  ! isspace (*l);  *w++ = *l++)  ;  *w = '\0';
    while (isspace (*l)) ++l;
    if (*word == 'G') return new Global_obj();
    else if (*word == 'r') {
        float cx, cy, len, wid;
        l = get_f (cx, l);  l = get_f (cy, l);
        l = get_f (len, l);  l = get_f (wid, l);
        return new Rect_obj (cx, cy, len, wid);
    }
    else if (*word == 'c') {
        float cx, cy, r;
        l = get_f (cx, l);  l = get_f (cy, l);  l = get_f (r, l);
        return new Circle_obj (cx, cy, r);
    }
    else {
        fprintf (stderr, "GeomObj: Unknown type: %s", line);
        return 0;
    }
}

// Simultaneously calculate sine and cosine, saving one trig call
void sincos (float x, float &s, float &c)
{
```

```
    s = sin (x);
    c = sqrt (1 - s * s);
    if (PI / 2 < x && x < 3 * PI / 2) c = -c;
}

// Find the (x,y,z) corrdinate mapped from (R,φ,θ)
void cart_map (float &x, float &y, float &z,
               float phi, float theta, float R_h, float R_v)
{
    float sp, cp, st, ct;

    //    phi_trig (sp, cp, phi);          theta_trig (st, ct, theta);
    sincos (phi, sp, cp);          sincos (theta, st, ct);

    x = R_h * cp * st; y = R_h * sp * st;  z = R_v * ct;
}

Coords *coords = 0;

Coords::Coords (void)
{
    Point_3D *p;
    Element *e;
    float phi, theta;
    float R_h, R_v;

    lattice->get_rad (R_h, R_v);
    lattice_coords = new Point_3D [ lattice->get_lattice_size() ] ;
    for (p = lattice_coords, e = lattice->lattice_start();
         e < lattice->lattice_end();  ++e, ++p) {
        lattice->angle_location (e, phi, theta);
        cart_map (p->x, p->y, p->z, phi, theta, R_h, R_v);
    }
}

// Find (φ,θ,R) coordinates from (x,y,z)

void surface_map (float x, float y, float z, float R_h, float R_v,
                  float &phi, float &theta)
{
    theta = acos (z / R_h);   phi = atan2 (y, x);
    theta /= PI;        theta = 1 - theta;
    if (phi < 0) phi += 2 * PI;
    phi /= 2 * PI;
}

// This will rotate a triplet by φ then θ, corresponding to a
// rotation of the North pole onto the (φ,θ) coordinate specified
void translate (float &x, float &y, float &z, float phi, float theta)
{
    float tx, ty, tz;
```

```
    float sp, cp, st, ct;
                                                                        100
    //  x' = R(φ,θ)x
    //  [x', y'] = | cos φ − sin φ||x|
    //             | sin φ cos φ||y|
    //    phi_trig (sp, cp, phi);
    sincos (phi, sp, cp);
    tx = cp * x − sp * y;    ty = cp * y + sp * x; tz = z;

    // now as above for [x, z] on θ
    //    theta_trig (st, ct, theta);
    sincos (theta, st, ct);                                             110
    x = ct * tx − st * tz;    z = ct * tz + st * tx; y = ty;
}

void GeomObj::operator() (void (*fn) (Element *))
{
    float phi_test, theta_test;
    Element *e;

    for (e = lattice->lattice_start();  e != lattice->lattice_end();  ++e) {
        if (e->is_dummy()) continue;                                    120
        lattice->angle_location (e, phi_test, theta_test);
        if (this->operator() (e)) fn (e);
    }
}

ostream& operator<< (ostream& os, GeomObj& g)
{
    Circle_obj *c;
    Splash_obj *s;
    switch (g.get_type()) {                                             130
        case 'c' :
            c = (Circle_obj *)&g;
            os << *c; break;
        case 'G' : os << "global"; break;
        case 's' :
            s = (Splash_obj *)&g;
            os << *s; break;
        default : os << "Unimplemented object"; break;
    }
    return os;                                                          140
}

ostream& operator<< (ostream& os, Circle_obj& c)
{
    int old_precision = os.precision();
    os << setprecision (3) <<
        "circle " << c.cx << " " << c.cy << " " << c.r <<
        setprecision (old_precision);
```

```
    return os;
}

ostream& operator<< (ostream& os, Splash_obj& s)
{
    int old_precision = os.precision();
    os << setprecision (3) <<
        "splash " << s.cx << " " << s.cy << " " <<
        s.r_min << " " << s.r_max <<
        setprecision (old_precision);
    return os;
}

// Rect_obj is not implemented in the current version of the simulator
// (4/24/98)
int Rect_obj::operator() (Element *e)
{
    float x, y, z;
    float pc, tc, phi_test, theta_test, phi_c, theta_c;
    float R = lattice->get_x_len() * lattice->get_scale() / (2 * PI);
    void draw_elem (Element *);
    Element *ce;

    if (! init) {
        setup (cx, cy);
        half_l = length / (2 * lattice->get_x_len() * lattice->get_scale());
        half_h = height / (2 * lattice->get_y_len() * lattice->get_scale());
        init = 1;
    }
    lattice->angle_location (e, phi_test, theta_test);
    ce = lattice->map (cx, cy);
    lattice->angle_location (ce, phi_test, theta_test);
    cart_map (x, y, z, phi_test, theta_test, R, R);
    translate (x, y, z,  1 - phi_c, 0.5 + theta_c);
    translate (x, y, z, 0.5, 1);
    surface_map (x, y, z, pc, tc, R, R);

    if (pc < 0.5 -half_l || pc > 0.5 + half_l ||
        tc < 0.5 -half_h || tc > 0.5 + half_h) return 0;
    return 1;
}
```

A.14 substrate.cc

Electrical "disease" is implemented, in general, as local variation in electrical properties of the tissue. Most of the mechanisms for doing this are implemented in the file *substrate.cc*. Most types of electrical inhomogeneity are implemented are GeomObjs (Sec. A.12) with specific electrical properties that are different from the rest of the substrate. The following types of inhomogeneities are implemented

Anisotropy A region of tissue in which the local propagation speed anisotropy ratio is different from the surrounding tissue. This is obviously not always associated with disease, since it is characteristic of healthy hearts.

Scars Regions of unexcitable tissue.

Diffuse Necrosis Regions of tissue in which a fraction of the element are "necrotic" (unexcitable).

APD Dispersion Regions of tissue in which there is variation of the action potential duration ("dispersion of refractoriness").

Infarctions Radially-symmetric regions of tissue in which the viability (the fraction of excitable elements) varies from 0 at the center to 1 at the extreme edge. This is designed to model an infarction with a "transition region" of partially-viable tissue.

A final type of inhomogeneity, "plaques," which are small patches of tissue with different properties are implemented in the file *plaques.cc* (section A.15).

```
#include <stdio.h>
#include <math.h>
#include <iostream.h>
#include "Element.h"
```

```
#include "Lattice.h"
#include "Params.h"
#include "GeomObj.h"
#include "logfile.h"
#include "Queue.h"
#include "Distribution.h"
#include "Neighbors.h"

GeomObj *get_geom_obj (char *&l);

Queue<GeomObj> scars;

void scar (char *l)
{
    scars += get_geom_obj (l);
}

void make_scar (Element *e)
{
    e->make_scar();
}

void do_scars (void)
{
    GeomObj *o;
    Element *e, *ledge, *hedge, *redge;
    float xmin, ymin, xmax, ymax;
    float x, y, dx, dy, r;

    for (o = scars.first(); o; o = scars.next())
        (*o) (make_scar);
}

struct aniso {
    float haxis, vaxis, rot, scatter;
    GeomObj *obj;
    aniso (float h, float v, float r, float s, GeomObj *o) :
        haxis (h), vaxis (v), rot (r), scatter (s), obj (o) { ; }
};

Queue<aniso> anisotropes;

void anisotropy (char *l)
{
    char word[32], *w, *line = l;
    float haxis = 1.0, vaxis = 1.0, rot = 0.0, scatter = 1.0;
    GeomObj *g;

    g = get_geom_obj (l);
    sscanf (l, "%f %f %f %f", &haxis, &vaxis, &rot, &scatter);
    anisotropes += new aniso (haxis, vaxis, rot, scatter, g);
```

```
}

static Hood_dist *a_hoods;

static void set_hood (Element *e) { e->set_hood_dist (a_hoods); }

void do_anisotropy (void)
{
    aniso *a;
    float r = rads[def_speed];

    for (a = anisotropes.first(); a; a = anisotropes.next()) {
        a_hoods = new Hood_dist (64, a->haxis * r, a->vaxis * r,
                                 a->scatter, round (a->rot * PI / 180.0));
        (*a->obj) (set_hood);
    }
}

static struct necr {
    float necr_frac;
    GeomObj *obj;
    Uniform *rand;
    necr (float n, GeomObj *g) : necr_frac (n), obj (g), rand (0) { ; }
    int operator() (void) {
        if (! rand) rand = new Uniform (0, 1.0, rand_gen);
        return (*rand)() < cur_necr->necr_frac;
    }
} *cur_necr;

Queue<necr> necs;

void necrosis (char *l)
{
    GeomObj *g = get_geom_obj (l);
    necs += new necr (atof (l), g);
}

static void set_necr (Element *e) {
    if ((*cur_necr) ()) e->make_scar();
}

void do_necrosis (void)
{
    for (cur_necr = necs.first(); cur_necr; cur_necr = necs.next())
        (*cur_necr->obj) (set_necr);
}

static struct apd_distr {
    GeomObj *obj;
    Normal *rand;
    float min, mean, sig;
```

```
    apd_distr (float m, float s, GeomObj *g, float low = 0) : obj (g),
        min (low), mean (m), sig (s), rand (0) { ; }
    float operator() (void) {
        float d;
        if (! rand) rand = new Normal (mean, sig * sig, rand_gen);
        while ((d = (*rand)()) < min) ;
        return d;
    }
} *cur_distr;

Queue<apd_distr> apds;

void dispersion (char *l)
{
    float m, s, min = 0;
    GeomObj *g = get_geom_obj (l);
    sscanf (l, "%f %f %f", &m, &s, &min);
    apds += new apd_distr (m, s, g, min);
}

static void set_apd (Element *e) {
    extern float time_step;

    e->set_base_ref (round ((*cur_distr)() / time_step));
}

void do_apd_distr (void)
{
    for (cur_distr = apds.first(); cur_distr; cur_distr = apds.next())
        (*cur_distr->obj) (set_apd);
}

static float r_lin (float r) { return r; }
static float r_quad (float r) { return r * r; }
static float r_cub (float r) { return r * r * r; }

struct infarct : public Splash_obj {
    Uniform *rand;
    float cx, cy, r_min, r_max;
    infarct (float x, float y, float min, float max, float (*f) (float)) :
        Splash_obj (x, y, min, max, f), rand (0)
        { ; }
    int operator() (Element *e) {
        float pe, te, level;
        if (! rand) rand = new Uniform (0, 1.0, rand_gen);

        if (Splash_obj::operator() (e, level)) return  (*rand)() < level;
        else return 0;
    }
};
```

```
static struct infarct *cur_infarct;

Queue<struct infarct> infarcts;                                          160

void infarct (char *l)
{
    float cx, cy, r_min, r_max;
    char name[32];
    float (*f) (float);
    sscanf (l, "%f %f %f %f %s", &cx, &cy, &r_min, &r_max, name);
    if (*name == 'q') f = r_quad;
    else if (*name == 'c') f = r_cub;
    else f = r_lin;                                                      170
    infarcts += new struct infarct (cx, cy, r_min, r_max, f);
}

static void set_infarct (Element *e)
{
    if ((*cur_infarct) (e)) e->make_scar();
}

void do_infarcts (void)
{                                                                        180
    Element *e;

    for (cur_infarct = infarcts.first(); cur_infarct;
         cur_infarct = infarcts.next())
        for (e = lattice->lattice_start(); e != lattice->lattice_end(); ++e)
            if (! e->is_dummy() && (*cur_infarct) (e)) e->make_scar();
}

void setup_substrate (void)
{                                                                        190
    Element *e;
    void apd_plaques (char *);
    void do_apd_plaques (void);

    new Param ("scar", scar);
    new Param ("anisotropy", anisotropy);
    new Param ("necrosis", necrosis);
    new Param ("apd_distr", dispersion);
    new Param ("infarct", infarct);
    new Param ("apd_plaques", apd_plaques);                              200

    get_params();

    do_scars();
    do_anisotropy();
    do_necrosis();
    do_apd_distr();
    do_infarcts();
```

```
    do_apd_plaques();
                                                                    210
    for (e = lattice->lattice_start(); e < lattice->lattice_end(); ++e)
        if (! e->is_dummy()) e->set_lsp();
}
```

A.15 plaque.cc

```
#include <stdio.h>
#include <math.h>
#include <iostream.h>
#include "Element.h"
#include "Lattice.h"
#include "Params.h"
#include "GeomObj.h"
#include "logfile.h"
#include "Queue.h"
#include "Distribution.h"
#include "Neighbors.h"
#include "Plaque.h"

void Plaque_distr::operator() (void)
{
    Element *e;
    int n;
    float r, scale = lattice->get_scale();
    float x, y;
    Hood_dist *get_hoods (int n, float h, float v, float s = 1.0,
                          int r1 = 0, int r2 = 0);
    Hood_dist *hood;
    Uniform rand (0, 1.0, rand_gen);
    Normal nrm (r_m, r_s * r_s, rand_gen);

    // calculate number of plaques needed to give specified coverage
    n = round (coverage / (area() / obj->area()));

    printf ("doing %d plaques.\n", n);
    while (--n >= 0) {              // loop over plaques
        do {                        // find "legal" plaque center
            x = rand(); y = rand();   // find (x,y) ∈ [0,1]
            e = lattice->nrm_map (x, y);
        } while (! (*obj) (e));     // reject if not in object specified
        // Now we've found a suitable element to be the center of a new plaque
        do {
            r = nrm();              // get a radius for the plaque
            r = round (r / dr) * dr; // discretize the radius
        } while (r <= 0);
        //      printf ("doing a plaque at %.2f cm.\n", r);
        hood = get_hoods (32, r / scale, r / scale, 1.5, 0, 0);
        (*fn) (0);      // tell the function it's about to get a new plaque
        e->traverse_neighborhood (fn, hood); // now do it.
    }
}
```

```
Queue<Plaque_distr> plaque_distrs;

static struct apd_plaque_distr {                                        50
    float apd_mean, apd_sigma;
    class Plaque_distr *distr;
    Normal *nrm;
    apd_plaque_distr (float m, float s, Plaque_distr *d) :
        apd_mean (m), apd_sigma (s), distr (d), nrm (0) { ; }
    float operator() (void) {
        if (! nrm) nrm = new Normal (apd_mean, apd_sigma * apd_sigma, rand_gen);
        return (*nrm)();
    }
} *cur_apd_plaque;                                                      60

Queue<apd_plaque_distr> apd_plaque_distrs;

int set_plaque (Element *e)
{
    static float apd;
    extern float time_step;

    if (! e) apd = (*cur_apd_plaque)();
    else e->set_base_ref (round (apd / time_step));                     70
    return 1;
}

void apd_plaques (char *l)
{
    float r_mean, r_sigma, apd_mean, apd_sigma, cov;
    GeomObj *g, *get_geom_obj (char *&l);
    g = get_geom_obj (l);
    sscanf (l, "%f %f %f %f %f",
            &r_mean, &r_sigma, &cov, &apd_mean, &apd_sigma);            80
    apd_plaque_distrs +=
        new apd_plaque_distr (apd_mean, apd_sigma,
                              new Plaque_distr (g, r_mean, r_sigma,
                                                cov, set_plaque));
}

void do_apd_plaques (void)
{
    for (cur_apd_plaque = apd_plaque_distrs.first();
         cur_apd_plaque;  cur_apd_plaque = apd_plaque_distrs.next())    90
        (*(cur_apd_plaque->distr))();
}
```

Bibliography

[1] Abildskov, JA and RL Lux. Mechanisms in simulated torsade de pointes. *Journal of Cardiovascular Electrophysiology*, 4(5):547–560, October 1993.

[2] Abildskov, JA and RL Lux. Cycle-length effects on the initiation of simulated torsade de pointes. *Journal of Electrocardiology*, 27(1):1–9, January 1994.

[3] Abildskov, JA and RL Lux. Entrainment of reentrant tachycardia in a computer model. *Journal of Electrocardiology*, 27(4):277–286, October 1994.

[4] Abildskov, JA and RL Lux. Spiral waves in a computer model of cardiac excitation. *Pacing and Clinical Electrophysiology*, 17(5 Pt 1):944–952, May 1994.

[5] Agladze, K, JP Keener, SC Muller and A Panfilov. Rotating spiral waves created by geometry. *Science*, 264:1746–1748, Jun 1994.

[6] Alferness, C, PV Bayly, W Krassawska, JP Daubert, WM Smith and RE Ideker. Strength-interval curves in canine myocardium at very short cycle lengths. *Pacing and Clinical Electrophysiology*, 17(Part I):876–881, May 1994.

[7] Allessie, MA, FIM Bonke and FJG Schopman. Circus movement in rabbit atrial muscle as a mechanism of tachycardia. II. The role of nonuniform recovery of excitability in the occurrence of unidirectional block as studied with multiple microelectrodes. *Circulation Research*, 29(2):168–176, August 1976.

[8] Allessie, MA, K Konigs, CJHJ Kirchhof and M Wijffels. Electrophysiologic mechanisms of perpetuation of atrial fibrillation. *The American Journal of Cardiology*, 77:10A–23A, January 1996.

[9] Heart and stroke facts: 1994 statistical supplement. American Heart Association, 1993.

[10] Heart and stroke facts. American Heart Association, 1994.

[11] Beeler, GW and H Reuter. Reconstruction of the action potential of ventricular myocardial fibres. *Journal of Physiology*, 268:177–210, 1977.

[12] Belk, P, AB Feldman, LM Rosenband, YB Chernyak and RJ Cohen. A novel mechanism of arrhythmogenesis in tissue with samll-scale spatial variation of refractory periods. Submitted for publication.

[13] Belousov, BP. *Sbornik Referatov po Radiatsionni Meditsine.* Medgiz, Moscow, 1958.

[14] Beneken, JEW. Some computer models in cardiovascular research. In: D. H. Bergel, ed., *Cardiovascular Fluid Dynamics*, chapter 6, pp. 173–223. Academic Press, London, New York, 1972.

[15] Bursac, N. Private Communication.

[16] Chernyak, YB, AB Feldman and RJ Cohen. Correspondence between discrete and continuous models of excitable media: Trigger waves. *Physical Review E*, 55:3215, March 1996.

[17] Chung, AH. *Programmable Stimulator System for Study of Cardiac Arrhythmias.* Master's thesis, Massachusetts Instintite of Technology, 1993.

[18] Cohen, JI, P Belk, D Sherman, AB Feldman and YB Chernyak. Mechanisms of sudden cardiac death: A computer simulation analysis. Project Submitted to Westinghouse Talent Search, November 1996.

[19] Cohen, RJ. The tippe top revisited. *American Journal of Physics*, 45:12–17, 1977.

[20] Courtemanche, M, L Glass and JP Keener. Instabilities of a propagating pulse in a ring of excitable media. *Physical Review Letters*, 70(14):2182–2185, April 1993.

[21] Davis, TL. *Teaching Physiology through Interactive Simulation of Hemodynamics.* Master's thesis, Massachusetts Institute of Technology, 1991.

[22] Feldman, AB. Private Communication.

[23] Feldman, AB. *Spiral Waves in a Discrete Model of Excitable Media.* Ph.D. thesis, Harvard University, 1997.

[24] Feldman, AB, P Belk, YB Chernyak and RJ Cohen. Speed curvature relations in an anisotropic medium. In preparation.

[25] Feldman, AB, YB Chernyak and RJ Cohen. On measuring the electrical diffusion constants and critical curvatures in myocardium. In preparation.

[26] Feldman, AB, YB Chernyak and RJ Cohen. Wavefront propagation in a discrete model of excitable media. *Physical Review E.* To appear.

[27] Feldman, AB, YB Chernyak and RJ Cohen. Spiral waves are stable in discrete element models of two-dimensional homogeneous excitable media. *International Journal of Bifurcation and Chaos*, June 1998. To appear.

[28] FitzHugh, R. Impulses and physiological states in theoretical models of nerve membrane. *Biophysical Journal*, 1:445, 1961.

[29] Frame, L and MB Simson. Oscillations of conduction, actional potential duration and refractoriness: A mechanism of spontaneous termination of reentrant tachycardias. *Circulation*, 78:1277–1287, 1988.

[30] francis X. Witkowski, LJ Leon, PA Penkoske, WR Giles, ML Spano, WL Ditto and AT Winfree. Spatiotemporal evolution of ventricular fibrillation. *Nature*, 392:78–82, 5 March 1998.

[31] Franz, MR, CD Swerdlo, LB Liem and J Schaefer. Cycle length dependence of human action potential duration in vivo. *Journal of Clinical Investigation*, 82:972–979, 1988.

[32] Frazier, DW, PD Wolf, JM Wharton, ASL Tang, WM Smith and RE Ideker. Stimulus-induced critical point. Mechanism for electrical initiation of reentry in normal canine myocardium. *Journal of Clinical Investigation*, 83:1039–1052, March 1989.

[33] Gerhardt, M, H Schuster and JJ Tyson. A cellular automaton model of excitable media including curvature and dispersion. *Science*, 247:1563–1566, 1990.

[34] Gettes, LS. Effect of ischemia on cardiac electrophysiology. In: H. A. Fozzard, ed., *The Heart and Cardiovascular System*, chapter 57. Raven Press, 1986.

[35] Gill, RM, RJ Sweeney and PR Reid. Refractory period extension during ventricular pacing at fibrillatory pacing rates. *Pacing and Clinical Electrophysiology*, 20:647–653, March 1997.

[36] Guevara, MR, G Ward and Shrier. Electrical alternans and period-doubling bifurcations. *Computers in Cardiology*, 11:167–170, 1984.

[37] Han, J and GK Moe. Nonuniform recovery of excitability in ventricular muscle. *Circulation Research*, 14:44, 1964.

[38] Hodgkin, AL and AF Huxley. A quantitative description of membrane current and its applications to conductionn and excitation in nerve. *Journal of Physiology*, 117:500–544, 1952.

[39] Ideker, RE, X Zhou and SB Knisley. Correlation among fibrillation, defibrillation, and cardiac pacing. *Pacing and Clinical Electrophysiology*, 18(Part II):512–525, March 1995.

[40] Ito, H and L Glass. Spiral breakup in a new model of discrete excitable media. *Physical Review Letters*, 66:671, 1991.

[41] Josephson, ME, P Zimetbaum, D Huang, R Sauberman, KM Monahan and CS Callans. Pathophysiologic substrate for sustained ventricular tachycardia in coronary artery disease. *Japanese Circulation Journal*, 61(6):459–466, June 1997.

[42] Karagueuzian, HS, SS Khan, K Hong, Y Kobayashi, T Denton, WJ Mandel and GA Diamond. Action potential alternans and irregular dynamics in quinidine-intoxicated ventricular muscle cells: Implications for ventricular proarrhythmia. *Circulation*, 87(5):1661–1672, May 1993.

[43] Katz, AM. *Phsiology of the Heart, Second Edition.* Raven Press, New York, 1992.

[44] Keener, JP. *Theory of the Heart*, chapter Wave Propagation in Myocardium, pp. 405–436. Springer-Verlag, New York, NY, 1991.

[45] Lammers, WJEP, C Kirchhof, FIM Bonke and MA Allessie. Vulnerability of rabbit atrium to reentry by hypoxia. role of inhomogeneity in conduction and wavelength. *American Journal of Physiology*, 262:H47–H55, 1992.

[46] Luo, CH and Y Rudy. A model of the ventricular cardiac action potential. depolarization, repolarization, and their interaction. *Circulation Research*, 68(6):1501–1526, June 1991.

[47] Luo, CH and Y Rudy. A dynamic model of the cardiac ventricular action potential. i. simulations of ionic currents and concentration changes. *Circulation Research*, 74(6):1071–1096, June 1994.

[48] Luo, CH and Y Rudy. A dynamic model of the cardiac ventricular action potential. ii. afterdepolarizations, triggered activity, and potentiation. *Circulation Research*, 74(6):1097–1113, June 1994.

[49] Markus, M and B Hess. Isotropic cellular automaton for modelling excitable media. *Nature*, 347:56–58, September 1990.

[50] Markus, M, M Krafczyk and B Hess. Randomized automata for isotropic modelling of two- and thre-dimensional waves and spatiotemporal chaos in excitable media. In: A. V. Holden, M. Markus and H. G. Othmer, eds., *Nonlinear Wave Processes in Excitable Media*, chapter 17, pp. 161–182. Plenum Press, New York, 1991.

[51] Merx, W, MS Yoon and J Han. The role of local disparity in conduction and recovery time on ventricular vulnerability to fibrillation. *American Heart Journal*, 94(5):603–610, November 1977.

[52] Moe, GK, WC Rheinboldt and JA Abildskov. A computer model of atrial fibrillation. *American Heart Journal*, 67(2):200–220, February 1964.

[53] Mudge, GH, Jr. *Manual of Electrocardiography.* Little Brown and Company, Boston, 2nd edition, 1986.

[54] Nagumo, JS, S Arimoto and S Yoshizawa. *Proceedings of IEEE*, 50:2061, 1962.

[55] Penrose, R. Pentaplexity. *Mathematical Intelligencer*, 2:32–37, 1979. Cited in [50].

[56] Pertsov, AM, EA Ermakova and EE Shnol. On the diffraction of autowaves. *Physica D*, 44(1-2):178–190, August 1990.

[57] Rosenband, LM. *The Study of Aberrant Conduction in Myocardial Tissue using a Finite-element Computer Simulation.* Bachelors's thesis, Massachusetts Institute of Technology, 1996.

[58] Rosenbaum, DS, LE Jackson, JM Smith, H Garan, JN Ruskin and RJ Cohen. Electrical alternans and vulnerability to ventricular arrhythmias. *The New England Journal of Medicine*, 330(4):235–241, January 1994.

[59] Russell, DC and MF Oliver. Ventricular refractoriness during acute myocardial ischaemia and its relationship to ventricular fibrillation. *Cardiovascular Research*, 12:221–227, 1978.

[60] Saxberg, BEH. *A Theoretical Model of the electrical Activity of the Heart.* Ph.D. thesis, Massachusetts Institute of Technology, June 1989.

[61] Schalij, MJ, WJ Lammers, PL Rensma and MA Allessie. Anisotropic conduction and reentry in perfused epicardium of rabbit left ventricle. *American Journal of Physiology*, 263(5 Pt 2):H1466–78, November 1992.

[62] Smith, JM, EA Clancy, CR Valeri, JN Ruskin and RJ Cohen. Electrical alternans and cardiac electrical instability. *Circulation*, 77(1):110–121, January 1988.

[63] Smith, JM and RJ Cohen. Simple finite-element model accounts for wide range of cardiac dysrhythmias. *Proceeding of the National Academy of Sciences*, 81:233–237, January 1984.

[64] Smith, JM, DT Kaplan and RJ Cohen. The physics of reentry and fibrillation. In: D. P. Zipes and J. Jalife, eds., *Cardiac Electrophysiology from Cell to Bedside*, chapter 25. W. B. Saunders Company, 1990.

[65] Spach, MS and ME Josephson. Initiating reentry: the role of nonuniform anisotropy in small circuits. *Journal of Cardiovascular Electrophysiology*, 5(2):182–209, February 1994.

[66] Starobin, JM, YI Zilberter, EM Rusnak and CF Starmer. Wavelet formation in excitable cardiac tissue: The role of wavefront-obstacle interaction in initiating high-frequency fibrillatory-like arrhythmias. *Biophysical Journal*, 70, February 1996.

[67] Sweeney, RJ, RM Gill and PR Ried. Arrhythmias/pacing: Refractory interval after transcardiac shocks during ventricular fibrillation. *Circulation*, 94(11):2947–2952, December 1996.

[68] Swerdlow, CD, LB Liem and MR Franz. Summation and inhibition by ultrarapid train pacing in the human ventricle. *Circulation*, 76(5):1101–1109, November 1987.

[69] Taccardi, B, RL Lux, PR Ershler, R MacLeod, TJ Dustman and N Ingebrigtsen. Anatomical architecture and electrical activitiy of the heart. *Acta Cardiological*, 52(2):91–105, 1997.

[70] Tan, HL, CJY Hou, MR Lauer and RJ Sung. Electrophysiologic mechanisms of the long qt interval syndromes and torsade de pointes. *Circulation*, 122(9):701–714, May 1995.

[71] Wanzhen, Z, L glass and A Shrier. Evolution of rhythms during periodic stimulation of embryonic chick heart cell aggregates. *Circulation Research*, 69(4):1022–1033, October 1991.

[72] Wei, D, O Okazaki, K Harumi, E Harasawa and H Hosaka. Comparative simulation of excitation and body surface electrocardiogram with isotropic and anisotropic computer heart models. *IEEE Transactions on Biomedical Engineering*, 42(4):343–357, April 1995.

[73] Yin, JZ. *Finite Element Model of Cardiac Electrical Conduction.* Ph.D. thesis, Massachusetts Institute of Technology, 1994.

[74] Zeldovich, YB. *The Mathematical theory of Combustion and Explosions.* Consultants Bureau, New York, NY, 1985.

[75] Zhabotinskii, M. *Biophysica*, 9(329), 1964.